Vivado Design Suite Tutorial:

Programming and Debugging

UG936 (v 2014.2) June 4, 2014

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Changes
04/07/2014	2014.1	Updates to the tutorials to reflect the 2014.1 Vivado software changes.
06/04/2014	2014.2	Updates to the tutorials to reflect the 2014.2 Vivado software changes.

Table of Contents

Revision History 2

Debugging in Vivado Tutorial 6

Introduction 6

Objectives 6

Getting Started 7

Setup Requirements 7

Software 7

Hardware 7

Tutorial Design Components 8

Board Support and Pinout Information 8

Design Files 9

Connecting the Boards and Cables 11

Lab 1: Using the Netlist Insertion Method for Debugging a Design 12

Introduction 12

Step 1: Creating a Project with the Vivado New Project Wizard 12

Step 2: Synthesizing the Design 13

Step 3: Probing and Adding Debug IP 15

Adding Debug Nets to the Project 15

Running the Set Up Debug Wizard 22

Step 4: Implementing and Generating Bitstream 24

Lab 2: Using the HDL Instantiation Method for Debugging a Design in Vivado 26

Introduction 26

Step 1: Creating a Project with the Vivado New Project Wizard 26

Step 2: Synthesize Implement and Generate Bitstream 28

Lab 3: Using a VIO Core for Debugging a Design in Vivado 30

Introduction 30

Step 1: Creating a Project with the Vivado New Project Wizard 31

Step 2: Synthesize, Implement, and Generate Bitstream 36

Lab 4: Using Synplify Pro Synthesis Tool and Vivado for Debugging a Design 38

Introduction 38

Step 1: Create a Synplify Pro Project 38

Step 2: Synthesize the Synplify Project 45
Step 3: Create EDIF Netlists for the Black Box Created in Synplify Pro 46
Step 4: Create a Post Synthesis Project in Vivado IDE 47
Step 5: Add (more) Debug Nets to the Project 50
Running the Set up Debug Wizard 51
Step 6: Implementing the Design and Generating the Bitstream 52
Using Vivado Logic Analyzer to Debug Hardware 53
Introduction 53
Step 1: Verifying Operation of the Sine Wave Generator 53
Setting Up 53
Using the Vivado Integrated Logic Analyzer 54
Verifying Sine Wave Activity 60
Displaying the Sine Wave 61
Correcting Display of the Sine Wave 62
Step 2: Debugging the Sine Wave Sequencer State Machine (Optional) 64
Sine Wave Sequencer State Machine Overview 64
Viewing the State Machine Glitch 65
Fixing the Signal Glitch and Verifying the Correct State Machine Behavior 67
Verifying the VIO Core Activity (Only applicable to Lab 3) 68
Lab 5: Using Vivado Serial Analyzer to Debug Serial Links 76
Introduction 76
Design Description 76
Step 1: Creating, Customizing, and Generating an IBERT Design 77
Step 2: Adding an IBERT core to the Vivado Project 78
Step 3: Synthesize, Implement and Generate Bitstream for the IBERT design 84
Step 4: Interact with the IBERT core using Serial I/O Analyzer 86
Lab 6: Using Vivado ILA core to Debug JTAG-AXI Transactions 102
Introduction 102
What is the JTAG to AXI Master IP core? 102
Key Features 102
Additional Documentation 102
Design Description 103
Step 1: Opening the JTAG to AXI Master IP Example Design and Configuring the AXI Interface Debug Connections 103
Step 2: Program the KC705 Board and Interact with the JTAG to AXI Master Core 121
Step 3: Using ILA 3.0 Advanced Trigger Feature to Trigger on an AXI Read Transaction 128

Legal Notices 133
Please Read: Important Legal Notices 133

Introduction

This document contains a set of tutorials designed to help you debug complex FPGA designs. The first four labs explain different kinds of debug flows that you can chose to use during the course of debug. These labs introduce the Vivado® debug methodology recommended to debug your FPGA designs. The labs describe the steps involved in taking a small RTL design and the multiple ways of inserting the Integrated Logic Analyzer (ILA) core to help debug the design. The fifth lab is for debugging high-speed serial I/O links in Vivado. The first four labs converge at the same point when connected to a target hardware board.

Example RTL designs are used to illustrate overall integration flows between Vivado logic analyzer, ILA 3.0, and Vivado Integrated Design Environment (IDE). In order to be successful using this tutorial, you should have some basic knowledge of Vivado Design Suite tool flow.

Objectives

These tutorials:

- Show you how to take advantage of integrated Vivado logic analyzer features in the Vivado design environment that make the debug process faster and simpler.
- Provide specifics on how to use the Vivado IDE and the Vivado logic analyzer to debug common problems in FPGA logic designs.
- Provide specifics on how to use the Vivado Serial I/O Analyzer to debug high-speed serial links.

After completing this tutorial, you will be able to:

- Validate and debug your design using the Vivado Integrated Design Environment (IDE) and the Integrated Logic Analyzer (ILA) core.
- Understand how to create an RTL project, probe your design, insert an ILA 3.0 core, and implement the design in the Vivado IDE.
- Generate and customize an IP core netlist in the Vivado IDE.
- Debug the design using Vivado logic analyzer in real-time, and iterate the design using the Vivado IDE and a KC705 Evaluation Kit Base Board that incorporates a Kintex®-7 device.
- Analyze high-speed serial links using the Serial I/O Analyzer.

Getting Started

Setup Requirements

Before you start this tutorial, make sure you have and understand the hardware and software components needed to perform the labs included in this tutorial as listed below.

Software

- Vivado Design Suite 2014.1

Hardware

- Kintex-7 FPGA KC705 Evaluation Kit Base Board
- Digilent Cable
- Two SMA (Sub-miniature version A) cables

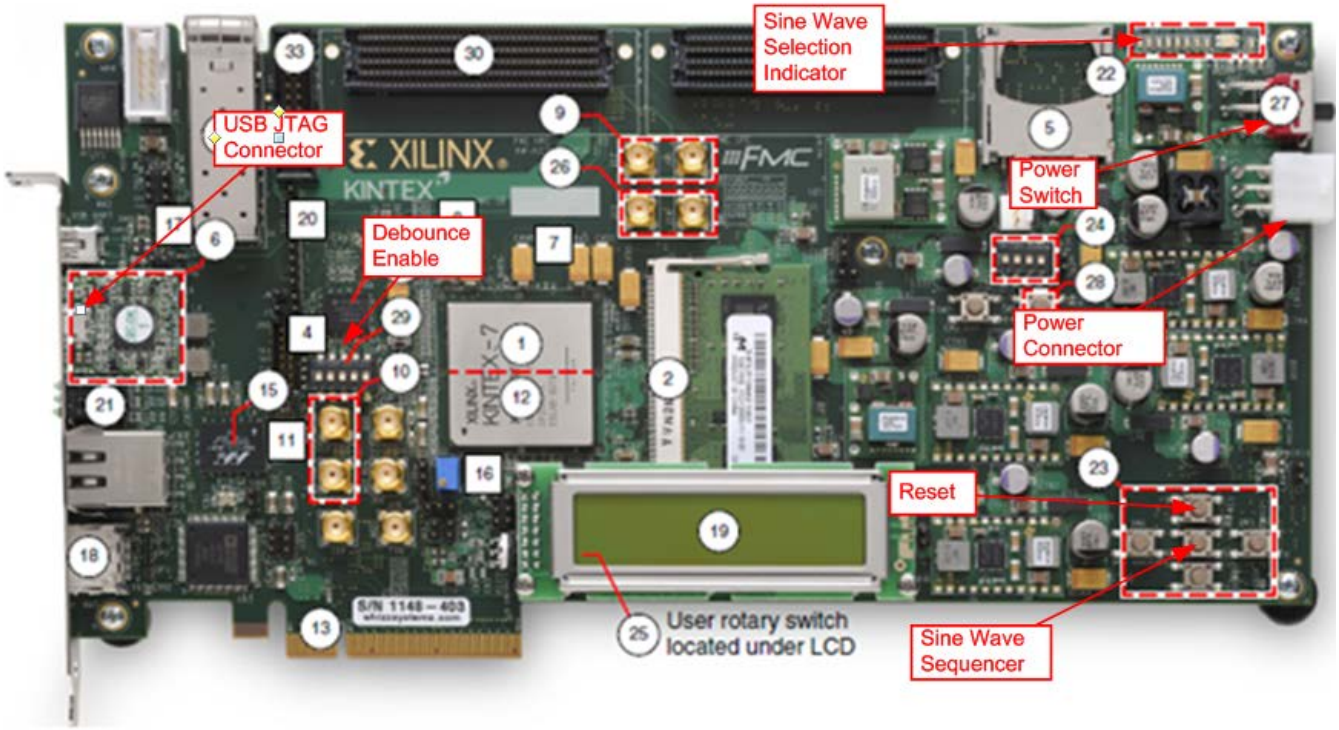


Figure 1: KC705 Board Showing Key Components

Tutorial Design Components

Labs 1 through 4 include:

- A simple control state machine
- Three sine wave generators using AXI-Streaming interface, native DDS Compiler
- Common push buttons (GPIO_BUTTON)
- DIP switches (GPIO_SWITCH)
- LED displays (GPIO_LED) VIO Core (Lab 3 only)

Push Button Switches: Serve as inputs to the de-bounce and control state machine circuits. Pushing a button generates a high-to-low transition pulse. Each generated output pulse is used as an input into the state machine.

DIP Switch: Enables or disables a de-bounce circuit.

De-bounce Circuit: In this example, when enabled, provides a clean pulse or transition from high to low. Eliminates a series of spikes or glitches when a button is pressed and released.

Sine Wave Sequencer State Machine: Captures and decodes input from the two push buttons. Provides sine wave selection and indicator circuits, sequencing among 00, 01, 10, and 11 (zero to three).

LED Displays: GPIO_LED_0 and GPIO_LED_1 display selection status from the state machine outputs, each of which represents a different sine wave frequency: high, medium, and low.

Lab5 includes:

- An IBERT core
- A top-level wrapper that instantiates the IBERT core.

Board Support and Pinout Information

Table 1: Pinout Information for the KC705 Board

Pinout Locations	Function	Pinout Locations
CLK_N	AD11	Clock
CLK_P	AD12	Clock
GPIO_BUTTONS[0]	AA12	Reset
GPIO_BUTTONS[1]	AG5	Sine Wave Sequencer
GPIO_SWITCH	Y28	De-bounce Circuit Selector
LEDS_n[0]	AB8	Sine Wave Selection[0]
LEDS_n[1]	AA8	Sine Wave Selection[1]
LEDS_n[2]	AC9	Reserved
LEDS_n[3]	AB9	Reserved

Design Files

1. In your `C:` drive, create a folder called `/Vivado_Debug`.
2. Find the tutorial source files at the following location: https://secure.xilinx.com/webreg/clickthrough.do?cid=358939&license=RefDesLicense&filename=ug936-vivado-tutorial-program-debug.zip

CAUTION! *The tutorial and design files may be updated or modified between software releases. You can download the latest version of the material from the Xilinx website.*

3. Unzip the tutorial source file to the `/Vivado_Debug` folder. There are five labs that use different methodologies for debugging your design. Select the appropriate lab and follow the steps to complete them

Lab 1: This lab walks you through the steps of marking nets for debug in HDL as well as the post-synthesis netlist (Netlist Insertion Method). Following are the required files:

- `debounce.vhd`
- `fsm.vhd`
- `rt.tcl`
- `sinegen.vhd`
- `sinegen_demo.vhd`
- `sine_high.xci`
- `sine_low.xci`
- `sine_mid.xci`
- `sinegen_demo_kc705.xdc`

Lab2: This lab goes over the details of marking nets for debug in the source HDL (HDL instantiation method) as well as instantiating an ILA core in the HDL. Following are the required files:

- `debounce.vhd`
- `fsm.vhd`
- `rt.tcl`
- `sinegen.vhd`
- `sinegen_demo_inst.vhd`
- `ila_0.xci`
- `sine_high.xci`
- `sine_low.xci`

- sine_mid.xci
- sinegen_demo_kc705.xdc

Lab 3: You can test your design even if the hardware is not physically accessible, using a VIO core. This lab will walk you over the steps of instantiating and customizing a VIO core that you will hook to the I/Os of the design. Following are the required files:

- debounce.vhd
- fsm.vhd
- sinegen.vhd
- sinegen_demo_inst_vio.vhd
- sine_high.xci
- sine_low.xci
- sine_mid.xci
- ila_0.xci
- sinegen_demo_kc705.xdc

Lab 4: Nets can also be marked for debug in a third-party synthesis tool using directives for the synthesis tool. This lab walks you through the steps of marking nets for debug in the Synplify tool and then using Vivado to perform the rest of the debug. Following are the required files:

- dds_compiler_v6_0_viv.edn
- dds_compiler_v6_0_viv_parameterized1.edn
- dds_compiler_v6_0_viv_parameterized3.edn
- debounce.vhd
- fsm.vhd
- sine_high.xci
- sine_low.xci
- sine_mid.xci
- sinegen.edn
- sinegen_synplify.vhd
- synplify_1.sdc
- sinegen_demo_kc705.xdc

Lab 5: Debug high-speed serial I/O links using the Vivado Serial I/O Analyzer. This lab uses the Vivado IP example design.

Lab 6: Using Vivado ILA core to debug JTAG-to-AXI transactions. This lab uses the Vivado IP example design.

Connecting the Boards and Cables

1. Connect the Digilent cable from the Digilent cable connector to a USB port on your computer.
2. Connect the two SMA cables (for lab 5 only) as follows:
   a. Connect one SMA cable from J19 (TXP) to J17 (RXP).
   b. Connect the other SMA cable from J20 (TXN) to J66 (RXN).

The relative locations of SMA cables on the board are shown in Figure 1: KC705 Board Showing Key Components.

Lab 1: Using the Netlist Insertion Method for Debugging a Design

Introduction

In this lab, you will mark signals for debug in the source HDL as well as the post synthesis netlist. Then you will create an ILA core and take the design through implementation. Finally, you will use Vivado® to connect to the KC705 target board and debug your design using Vivado Integrated Logic Analyzer.

Step 1: Creating a Project with the Vivado New Project Wizard

To create a project, use the New Project wizard to name the project, to add RTL source files and constraints, and to specify the target device.

1. Invoke the Vivado IDE.
2. In the **Getting Started** screen, click **Create New Project** to start the New Project wizard. Click **Next**.
3. In the **Project Name** screen, name the new project **proj_netlist** and provide the project location (`C:/Vivado_Debug`). Ensure that **Create Project Subdirectory** is selected and click **Next**.
4. In the **Project Type screen**, specify the **Type of Project** to create as **RTL Project**. Click **Next**.
5. In the **Add Sources** screen:
   a. Set **Target Language** to **VHDL**.
   b. Click the **Add Files** button.
   c. In the **Add Source Files** dialog box, navigate to the `/src/Lab 1 directory`.
   d. Select all VHD source files, and click **OK**.
   e. Verify that the files are added, and **Copy Sources into Project** is selected. Click **Next**.

7. In the **Add Existing IP (optional)** dialog box:
   a. Click the **Add Files** button.
   b. In the **Add Configurable IP** dialog box, navigate to the `/src/lab1` directory.
   c. Select all XCI source files, and click **OK**.
   d. Verify that the files are added and **Copy Sources into Project** is selected. Click **Next**.
8. In the **Add Constraints (optional)** dialog box, the provided XDC file, `sinegen_demo_kc705.xdc`, should automatically appear in the main window. Click **Next**.
9. In the **Default Part** dialog box, specify the **xc7k325tffg900-2** part for the KC705 platform. You can also select **Boards** and then select **Kintex-7 KC705 Evaluation Platform**. Click **Next**.
10. Review the **New Project Summary** page. Verify that the data appears as expected, per the steps above, and click **Finish**.

    Note: *It could take a moment for the project to initialize.*

Step 2: Synthesizing the Design

1. In the Project Manager, click **Project Settings** as shown in the following figure.

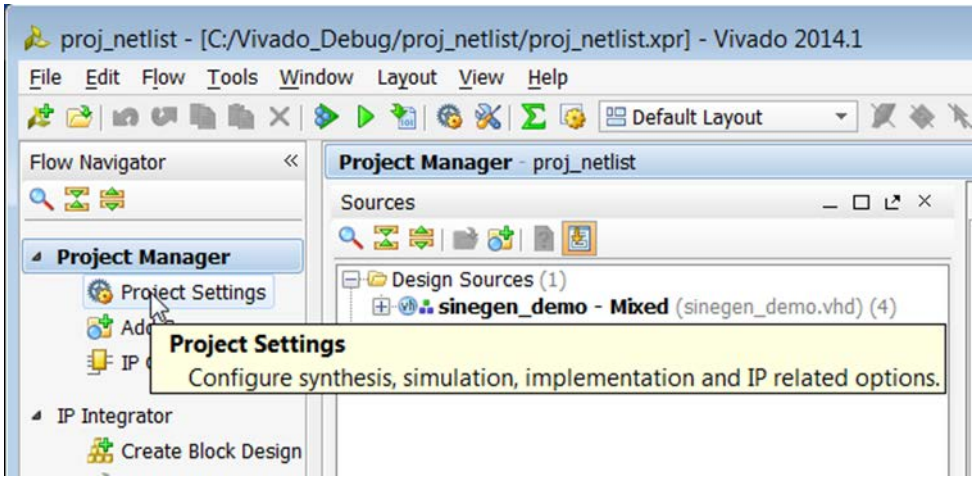


Figure 2: Configuring the Project Settings

3. In the **Project Settings** dialog box select **Synthesis** from the left pane and change **flatten_hierarchy option** to **none** as shown in shown in the following figure. Click **OK**.

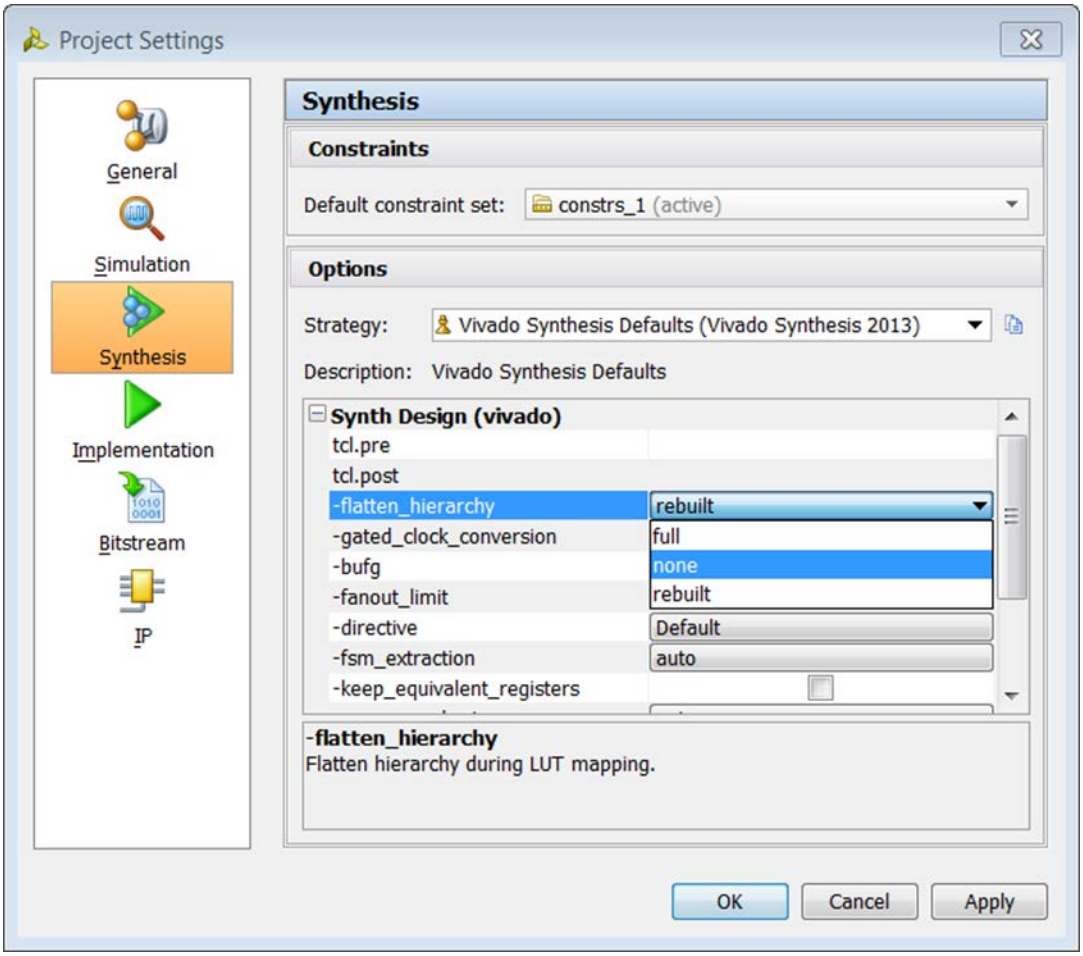


Figure 3: The Project Settings dialog box

IMPORTANT: *The reason for changing this setting to **none** is to prevent the synthesis tool from performing any boundary optimizations for this tutorial.*

4. In the Vivado Flow Navigator, expand the **Synthesis** drop-down list, and click **Run Synthesis**.

   Note: *When synthesis runs, a progress indicator appears, showing that synthesis is occurring. This could take a few minutes.*

6. In the **Synthesis Completed** dialog box, click **Cancel** as shown in the following figure. You will implement the design later.

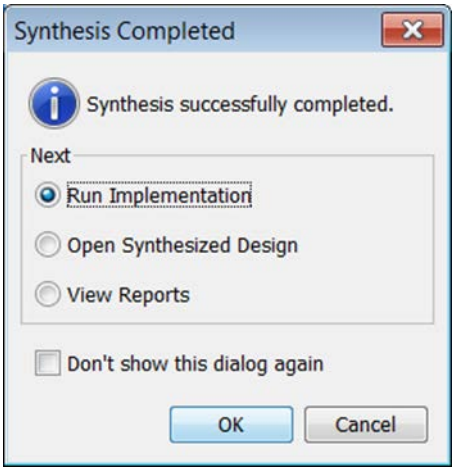


Figure 4: Synthesis Completed dialog box

Step 3: Probing and Adding Debug IP

To add a Vivado ILA 2.0 core to the design, take advantage of the integrated flows between the Vivado IDE and Vivado logic analyzer.

In this step, you will accomplish the following tasks:

- Add debug nets to the project.
- Run the Set Up Debug wizard.
- Implement and open the design.
- Generate the bitstream.

Adding Debug Nets to the Project

Following are some examples of how to add debug nets using the Vivado IDE:

- Add mark_debug attribute to the target XDC file.

```
set_property mark_debug true [get_nets sine*]
```

IMPORTANT: Use these attributes in synthesized designs only. Do not use then with pre-synthesis or elaborated design netlists.

- Add mark_debug attribute to HDL files.

  VHDL

```
attribute mark_debug : string;
attribute keep : string;
attribute mark_debug of sine     : signal is "true";
attribute mark_debug of sineSel  : signal is "true";
```

  Verilog

```
(* mark_debug = "true" *) wire sine;
(* mark_debug = "true" *) wire sineSel;
```

- Right-click and select **Mark Debug** or **Unmark Debug** on Synthesis netlist.
- Use a Tcl prompt to set the mark_debug attribute. For example:

```
set mark_debug true [get_nets sine*]
```

  This applies the mark_debug on the current, open netlist.

In the following steps, you learn how to add debug nets to HDL files and the synthesized design using Vivado IDE.

1. In the Flow Navigator under the **Synthesis** drop-down list, click **Open Synthesized Design** as shown in the following figure.

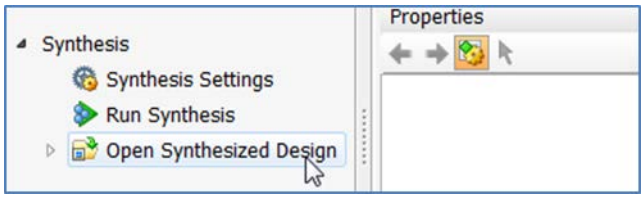


Figure 5: Open Synthesized Design

   Following are some examples alternate methods of adding debug nets using the Vivado IDE:

   o Add mark_debug attribute to the target XDC file.

```
set_property mark_debug true [get_nets sine*]
```

IMPORTANT: Use these attributes in synthesized designs only. Do not use then with pre-synthesis or elaborated design netlists.

- Add mark_debug attribute to HDL files.

  VHDL

```
attribute mark_debug : string;
attribute keep : string;
attribute mark_debug of sine     : signal is "true";
attribute mark_debug of sineSel  : signal is "true";
```

  Verilog

```
(* mark_debug = "true" *) wire sine;
(* mark_debug = "true" *) wire sineSel;
```

- Right-click and select **Mark Debug** or **Unmark Debug** on Synthesis netlist.
- Use a Tcl prompt to set the mark_debug attribute. For example:

```
set mark_debug true [get_nets sine*]
```

  This applies the mark_debug on the current, open netlist.

TIP: *Before proceeding, make sure that the Flow Navigator on the left panel is enabled. Use* ***Ctrl-Q*** *to toggle it off and on.*

---

2. In the Window menu, select **Layout > Debug**. When the **Debug** window opens. Click the window if it is not already selected.
3. Expand **Unassigned Debug Nets** folder. The following figure shows those debug nets that were tagged in `sinegen_demo.vhd` with mark_debug attributes, as shown in Figure 7.

```
-- Add mark_debug attributes to show debug nets in the synthesized netlist
attribute mark_debug : string;
attribute mark_debug of GPIO_BUTTONS_db : signal is "true";
attribute mark_debug of GPIO_BUTTONS_dly : signal is "true";
attribute mark_debug of GPIO_BUTTONS_re : signal is "true";

component sinegen
  port
  (
    clk   : in    std_logic;
    reset : in    std_logic;
    sel   : in    std_logic_vector(1 downto 0);
    sine  : out   std_logic_vector(19 downto 0)
  );
end component;
```

Figure 6: VHDL Example Using MARK_DEBUG Attributes

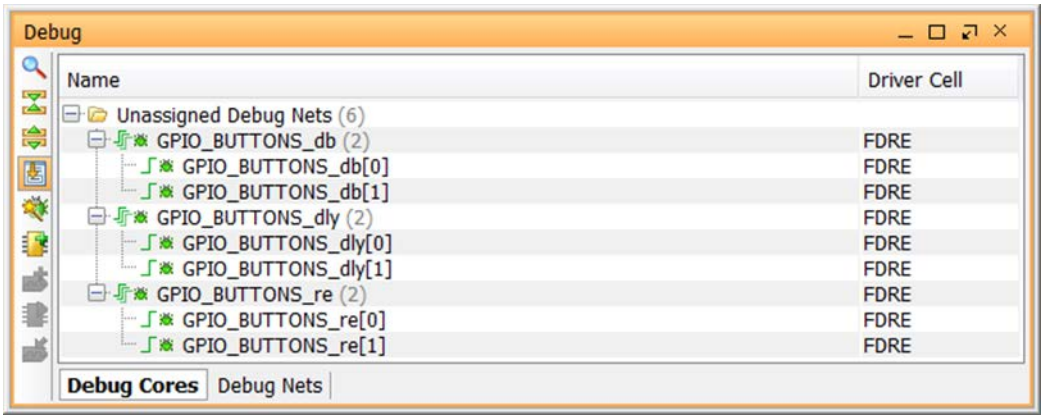


Figure 7: Unassigned Debug Nets Post-synthesis

4. Select the **Netlist** tab and expand **Nets**. Select the following nets for debugging shown in the following figure.
   - `GPIO_BUTTONS_IBUF[0]` and `GPIO_BUTTONS_IBUF[1]` - Nets folder under the top-level hierarchy
   - `sel(2)` - Nets folder under the U_SINEGEN hierarchy
   - `sine(20)`- Nets folder under the U_SINEGEN hierarchy

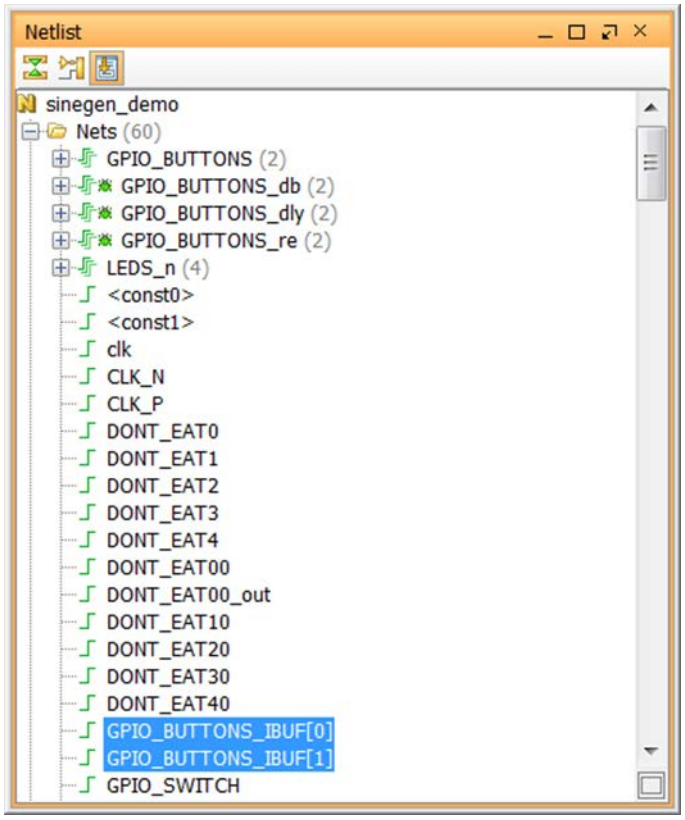


Figure 8: Add nets for debug from the synthesized netlist

***Note**: These signals represent the significant behavior of this design and are used to verify and debug the design in subsequent steps.*

6. Right-click the selected nets and select **Mark Debug** as shown in the following figure.

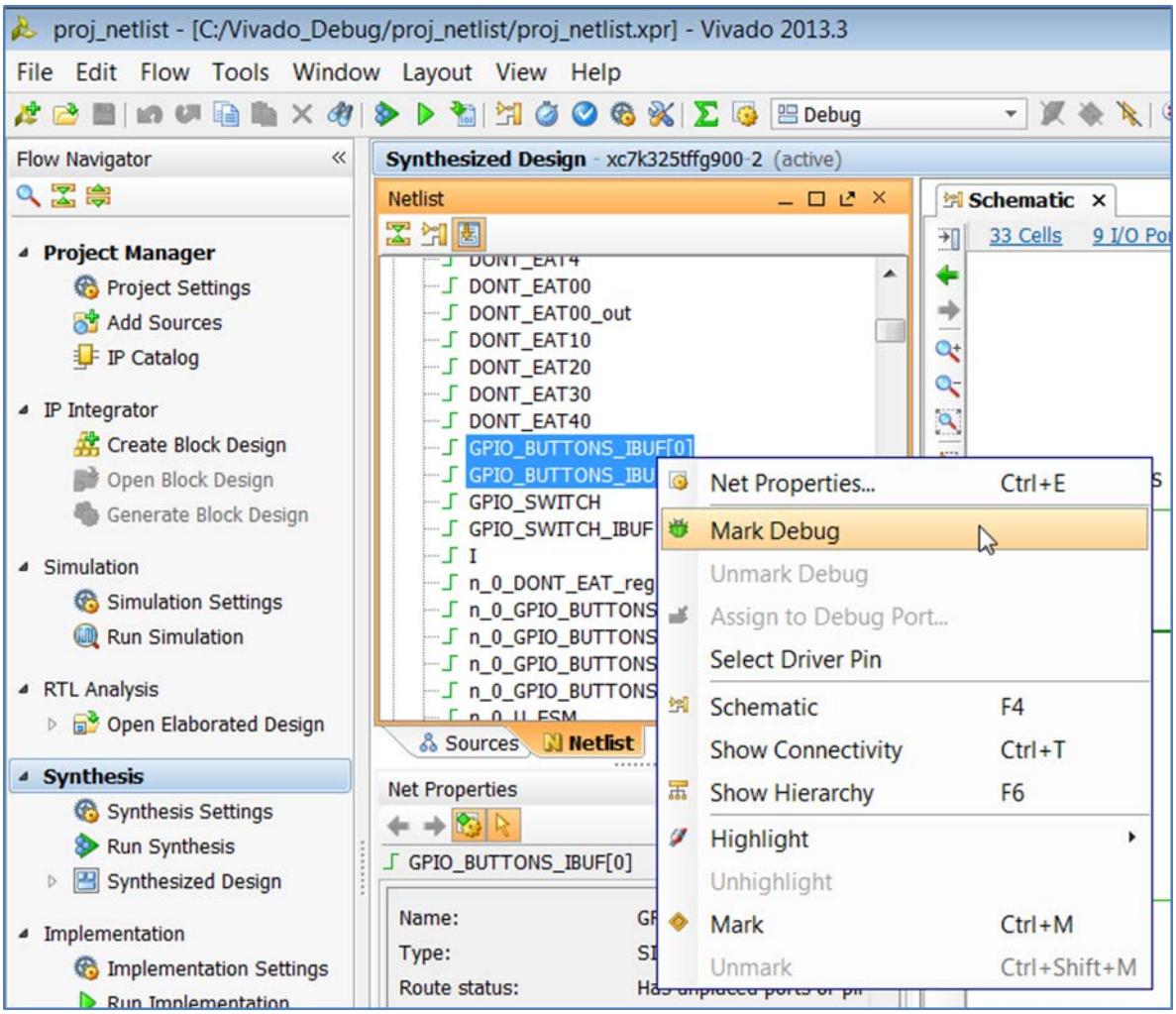


Figure 9: Adding Nets from the Netlist Tab

TIP: *In the **Debug** window, you can see the unassigned nets you just selected. In the **Netlist** window, you can also see the green bug icon next to each scalar or bus, which indicates that a net has the attribute* `mark_debug = true` *as shown the following two figures.*

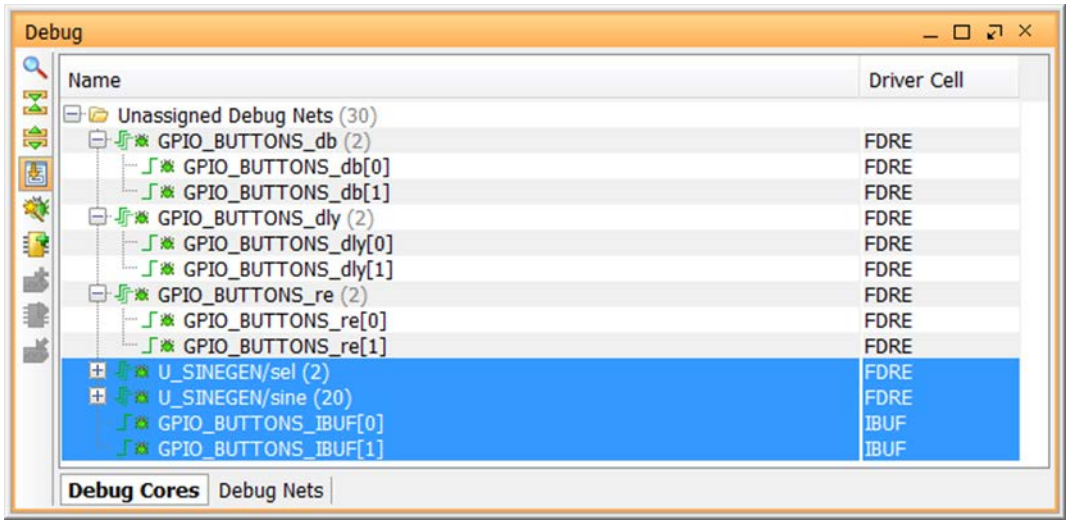


Figure 10: Newly added nets for debug from the synthesized netlist

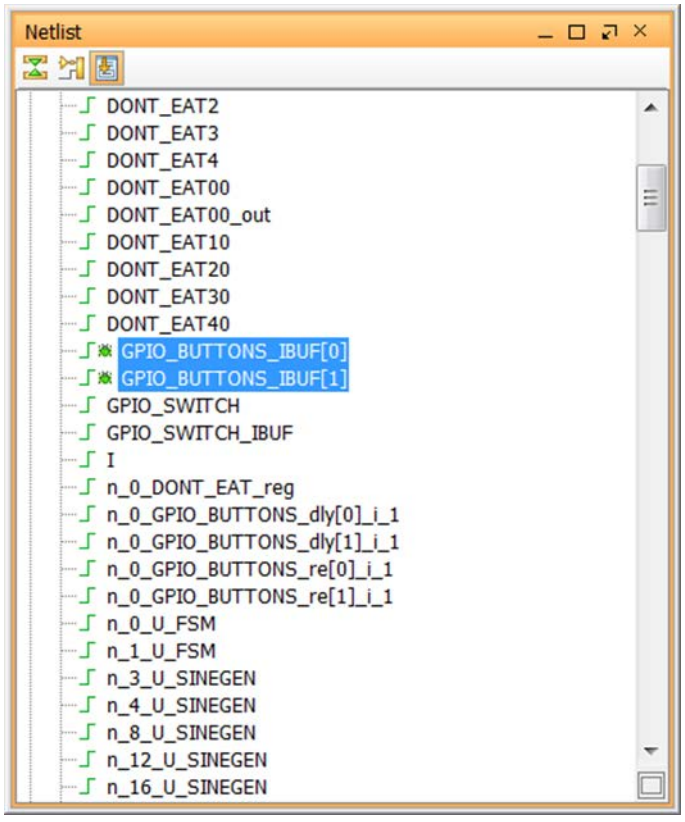


Figure 11: Netlist view of nets marked for debug

Running the Set Up Debug Wizard

7. From the **Debug** window or **Tools** menu, select **Set Up** Debug. The **Set Up Debug** wizard opens.

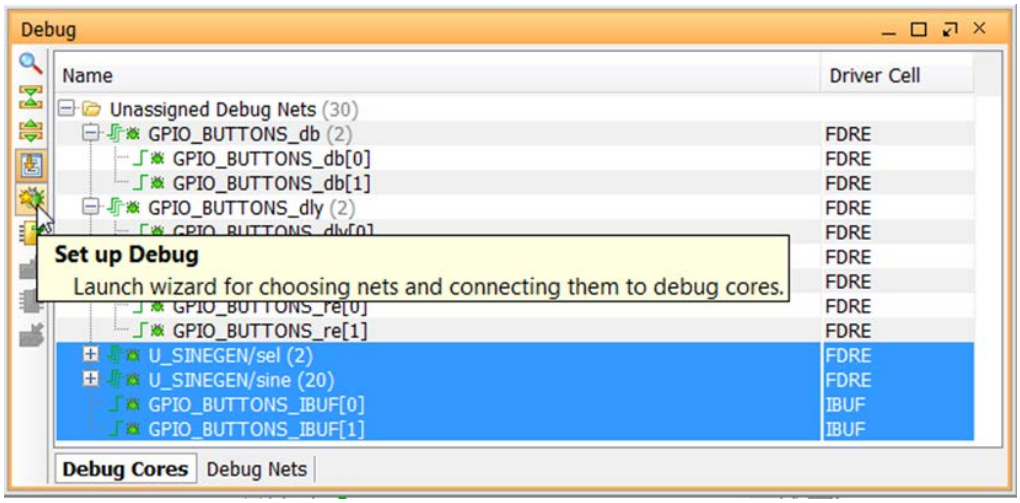


Figure 12: Launching the Set up Debug wizard

8. When the **Set up Debug** wizard opens, click **Next.**

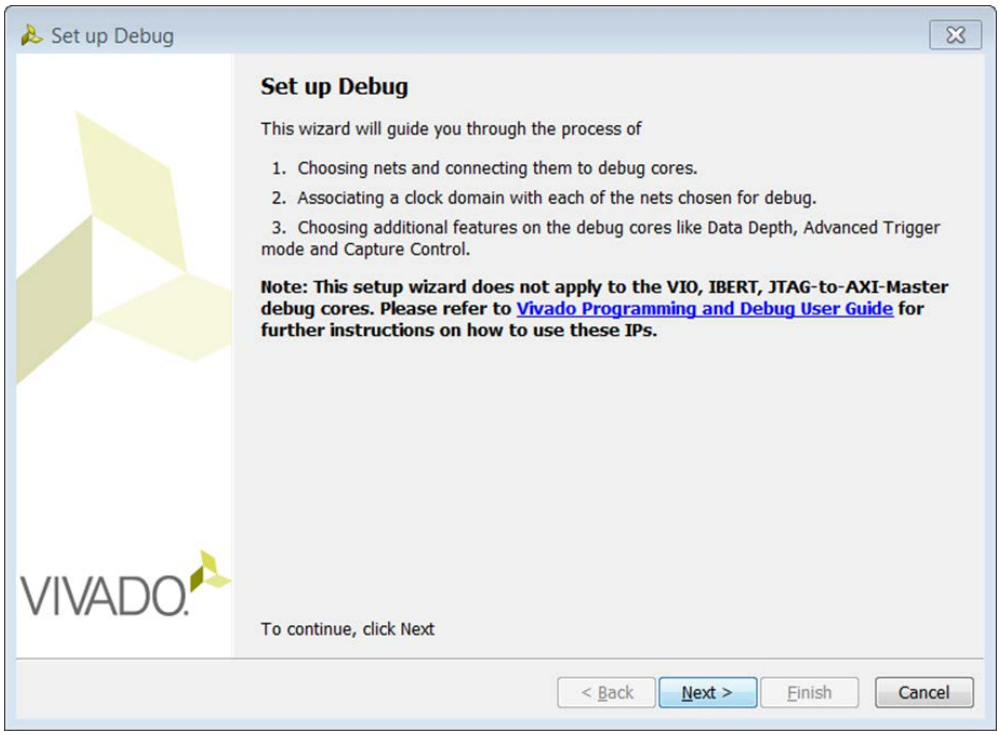

Set up Debug
Set up Debug
This wizard will guide you through the process of
1. Choosing nets and connecting them to debug cores.
2. Associating a clock domain with each of the nets chosen for debug.
3. Choosing additional features on the debug cores like Data Depth, Advanced Trigger mode and Capture Control.
Note: This setup wizard does not apply to the VIO, IBERT, JTAG-to-AXI-Master debug cores. Please refer to Vivado Programming and Debug User Guide for further instructions on how to use these IPs.
VIVADO
To continue, click Next
< Back
Next >
Finish
Cancel

Figure 13: Set up Debug dialog box

9. In the **Specify Nets to Debug** page, ensure that all the nets have been added for debug and click **Next**.

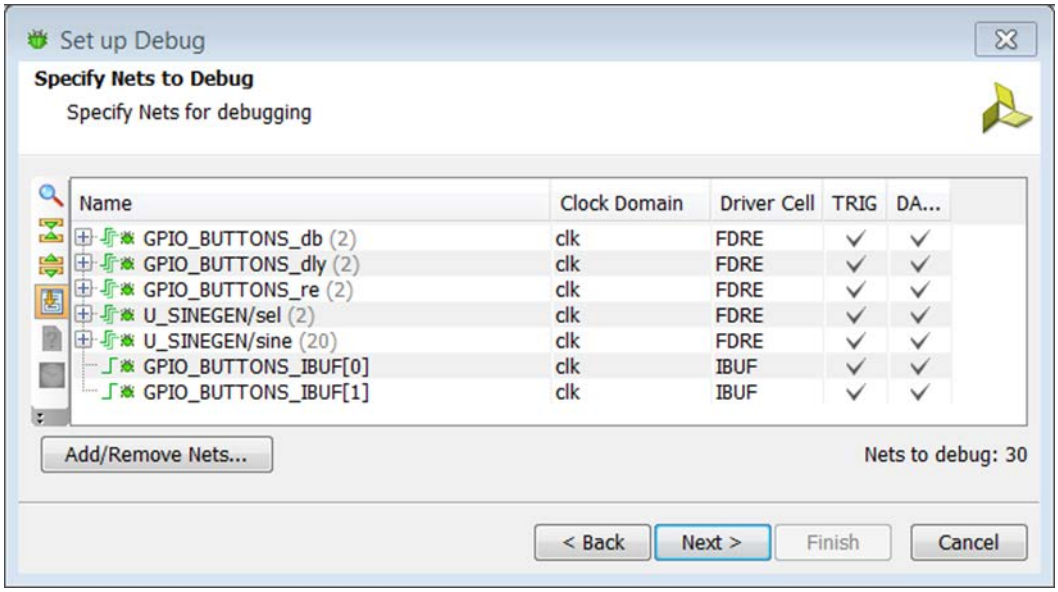


Figure 14: Specify Nets to Debug

10. In the **ILA General Options** page, go to the **Trigger and Storage Settings** section. Select both the **Capture Control** and **Advanced Trigger** settings. Click **Next**.

11. In the **Setup Debug Summary** page, make sure that all the information is correct and as expected. Click **Finish**.

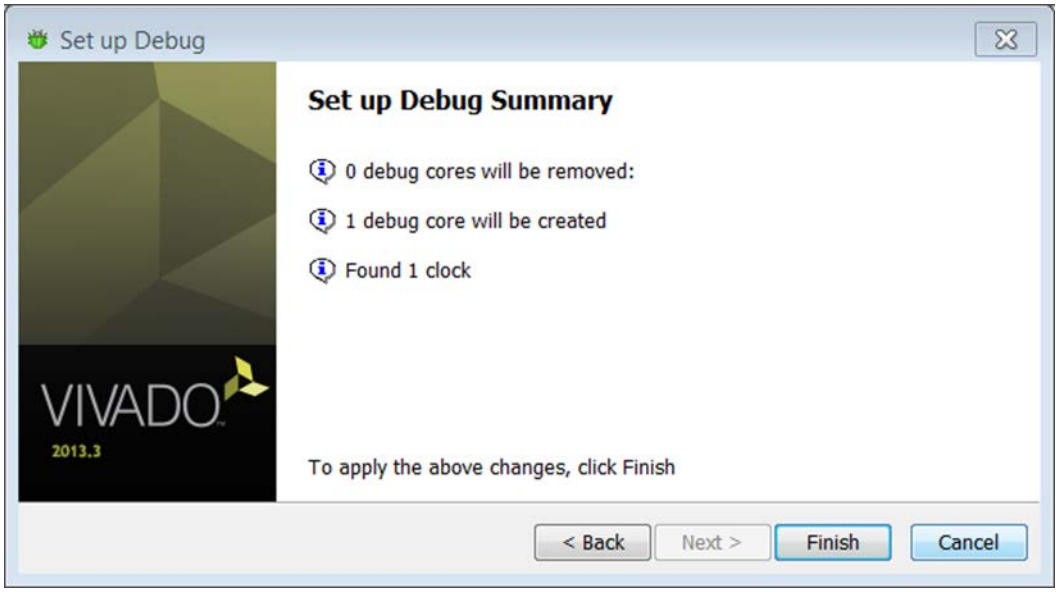


Figure 15: Set up Debug Summary

Step 4: Implementing and Generating Bitstream

1. Click **Generate Bitstream** from the **Program and Debug** drop-down list in the Flow Navigator.

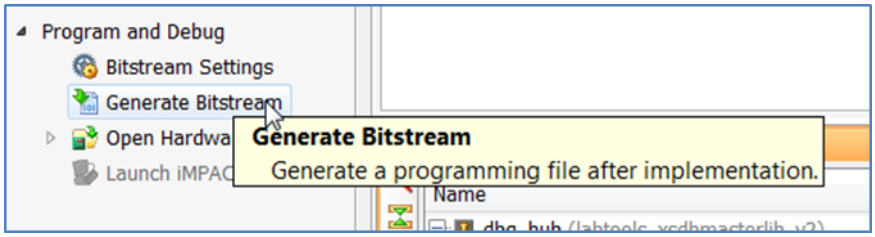


Figure 16: Implement design and Generate Bitstream

2. In the **Save Project** dialog box click **Save**. This applies the mark_debug attributes on the newly marked nets. You can see those constraints can by inspecting `the sinegen_demo_kc705.xdc` file.
3. When the **No Implementation Results Available** dialog box pops up. Click **Yes**.
4. When the bitstream generation completes, the **Bitstream Generation Completed** dialog box pops up. Click **OK**.

5. In the dialog box asking to close synthesized design before opening implemented design. Click **Yes**.
6. In the Implementation is **Out-of-date** dialog box, click **Yes**.
7. In the Flow Navigator, under **Implementation**, expand the **Implemented Design** drop-down list and select **Report Timing Summary**.

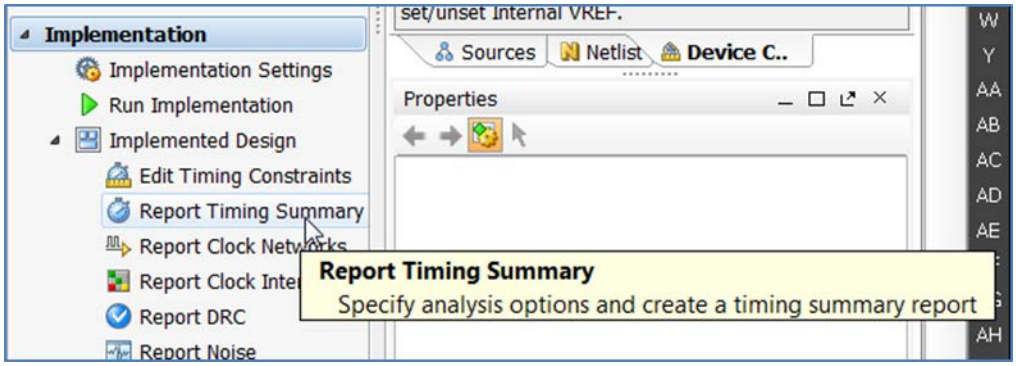


Figure 17: Analyze Timing Results

8. In the **Report Timing Summary** dialog box. Click **OK**.
9. Ensure that all the specified timing constraints are met.

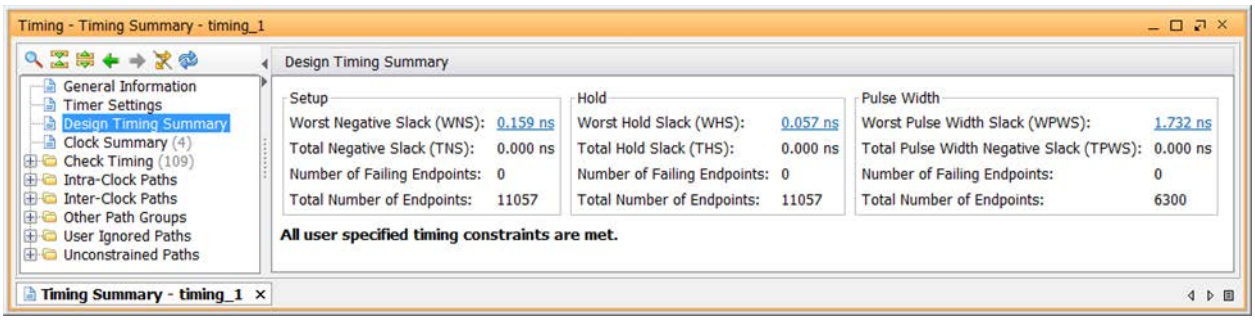

Timing - Timing Summary - timing_1
General Information
Timer Settings
Design Timing Summary
Clock Summary (4)
Check Timing (109)
Intra-Clock Paths
Inter-Clock Paths
Other Path Groups
User Ignored Paths
Unconstrained Paths
Design Timing Summary
Setup
Worst Negative Slack (WNS): 0.159 ns
Total Negative Slack (TNS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 11057
Hold
Worst Hold Slack (WHS): 0.057 ns
Total Hold Slack (THS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 11057
Pulse Width
Worst Pulse Width Slack (WPWS): 1.732 ns
Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 6300
All user specified timing constraints are met.
Timing Summary - timing_1

Figure 18: View the Timing Summary Report

10. Proceed to Using Vivado® Logic Analyzer to Debug Hardware to complete the rest of the steps for debugging the design.

Lab 2: Using the HDL Instantiation Method for Debugging a Design in Vivado

Introduction

The HDL Instantiation method is one of the two methods supported in Vivado® Debug Probing. For this flow, you will generate an ILA 3.0 IP using the Vivado IP Catalog and instantiate the core in a design manually as you would with any other IP.

Step 1: Creating a Project with the Vivado New Project Wizard

To create a project, use the New Project wizard to name the project, to add RTL source files and constraints, and to specify the target device.

1. Invoke the Vivado IDE.
2. In the **Getting Started** page, click **Create New Project** to start the New Project wizard. Click **Next**.
3. In the **Project Name** page, name the new project **proj_hdl** and provide the project location (`C:/Vivado_Debug`). Ensure that **Create Project Subdirectory** is selected. Click **Next**.
4. In the **Project Type** page, specify the **Type of Project** to create as **RTL Project**. Click **Next**.
5. In the **Add Sources** page:
   a. Set **Target Language** to **VHDL**.
   b. Click **Add Files**.
   c. In the **Add Source Files** dialog box, navigate to the `/src/Lab2` directory.
   d. Select all VHD source files, and click **OK**.
   e. Verify that the files are added, and **Copy Sources into Project** is selected. Click **Next**.
6. 6. In the **Add Existing IP (optional)** page:
   a. Click **Add Files**.
   b. In the **Add Configurable IP** page, navigate to the `/src` directory.

c. Select all XCI source files, and click **OK**.

d. Verify that the files are added, and **Copy Sources into Project** is selected. Click **Next**.

7. In the **Add Constraints (optional)** page, the provided XDC file, `sinegen_demo_kc705.xdc` should automatically appear in the main window. Click **Next**.
8. In the **Default Part** page, specify the **xc7k325tffg900-2** part for the KC705 platform. You can also select **Boards** and then select **Kintex-7 KC705 Evaluation Platform**. Click **Next**.
9. Review the **New Project Summary** page. Verify that the data appears as expected, per the steps above. Click **Finish**.
10. In the **Sources** window in Vivado IDE, expand **sinegen_demo_inst** to see the source files for this lab. Note that **ila_0** core has been added to the project.

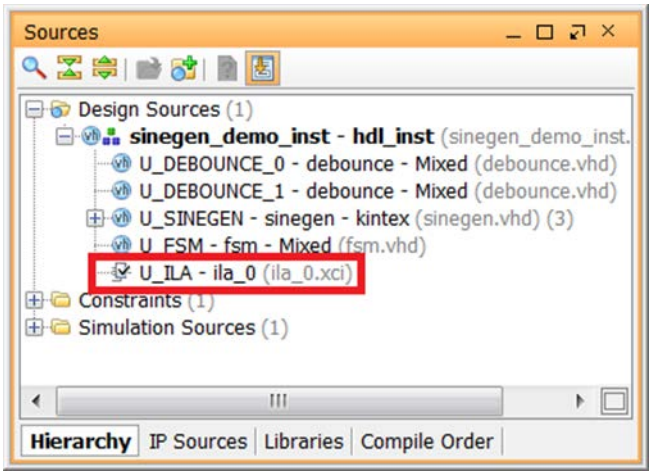


Figure 19: ILA instantiation in HDL

Double-click the **sinegen_demo_inst.vhd** file to open it and inspect the instantiation and port mapping of the ILA core in the HDL code. Note that attributes have been placed in the source file to preserve the net names.

```
-- Attributes for keeping the net names preserved
attribute keep : string;
attribute keep of GPIO_BUTTONS_db : signal is "true";
attribute keep of GPIO_BUTTONS_dly : signal is "true";
attribute keep of GPIO_BUTTONS_re : signal is "true";
attribute keep of sineSel : signal is "true";
attribute keep of sine : signal is "true";
```

Figure 20: Using "keep" attribute to preserve net names

```
------------------------------------------------------------
-- ILA
------------------------------------------------------------
U_ILA : ila_0
  port map
  (
    CLK => clk,
    PROBE0 => sineSel,
    PROBE1 => sine,
    PROBE2 => GPIO_BUTTONS_db,
    PROBE3 => GPIO_BUTTONS_re,
    PROBE4 => GPIO_BUTTONS_dly,
    PROBE5 => GPIO_BUTTONS
  );
```

Figure 21: Hook signals that need to be debugged in the ILA

Step 2: Synthesize Implement and Generate Bitstream

1. From the **Program and Debug** drop-down list, in Flow Navigator, click **Generate Bitstream**. This will synthesize, implement and generate a bitstream for the design.

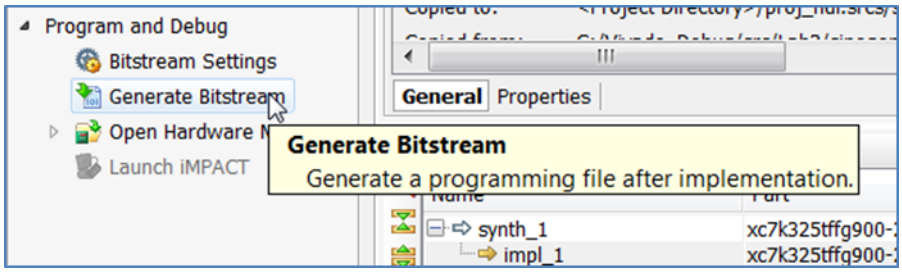


Figure 22: Generate Bitstream

2. The **No Implementation Results Available** dialog box appears. Click **Yes**.
3. After bitstream generation completes, the **Bitstream Generation Completed** dialog box appears. **Open Implemented Design** is selected by default. Click **OK**.

4. From the Implementation drop-down list, expand **Implemented Design** and select **Report Timing Summary**.

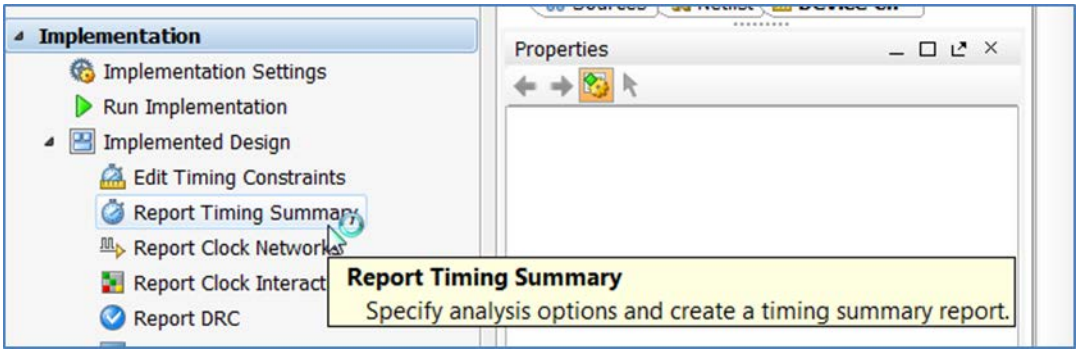


Figure 23: Report Timing Summary

5. In the **Report Timing Summary** dialog box. Make sure that all timing constraints are met. Click **OK.**

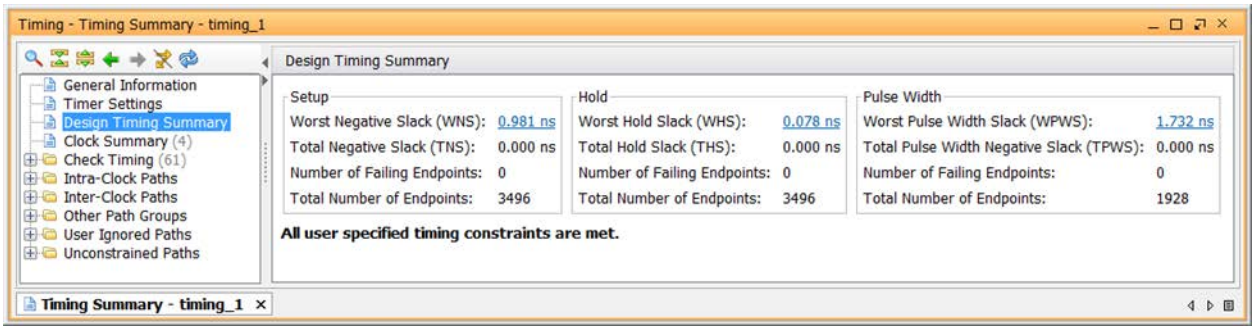

Timing - Timing Summary - timing_1
General Information
Timer Settings
Design Timing Summary
Clock Summary (4)
Check Timing (61)
Intra-Clock Paths
Inter-Clock Paths
Other Path Groups
User Ignored Paths
Unconstrained Paths
Design Timing Summary
Setup
Worst Negative Slack (WNS): 0.981 ns
Total Negative Slack (TNS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 3496
Hold
Worst Hold Slack (WHS): 0.078 ns
Total Hold Slack (THS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 3496
Pulse Width
Worst Pulse Width Slack (WPWS): 1.732 ns
Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 1928
All user specified timing constraints are met.
Timing Summary - timing_1

Figure 24: Review Timing Summary

6. Proceed to Using Vivado® Logic Analyzer to Debug Hardware to complete the rest of this lab.

Lab 3: Using a VIO Core for Debugging a Design in Vivado

Introduction

The Virtual Input/Output (VIO) core is a customizable core that can both monitor and drive internal FPGA signals in real time. The number and width of the input and output ports are customizable in size to interface with the FPGA design. Because the VIO core is synchronous to the design being monitored and/or driven, all design clock constraints that are applied to your design are also applied to the components inside the VIO core. Run time interaction with this core requires the use of the Vivado® logic analyzer feature. The following figure is a block diagram of the new VIO 3.0 core.

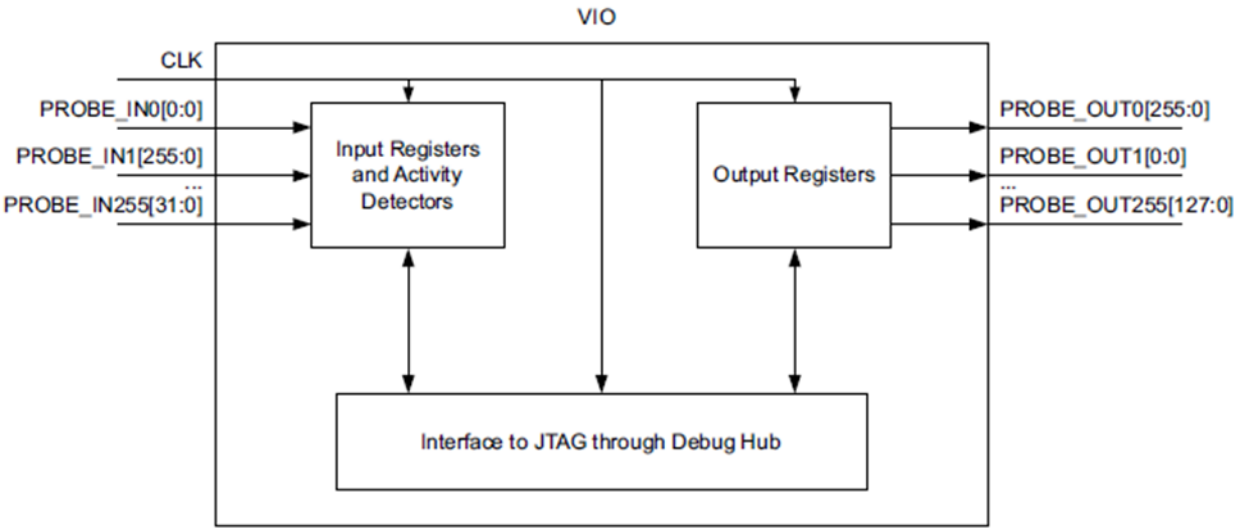


Figure 25: VIO Block Diagram

This lab will walk you through the steps of instantiating and configuring a VIO core and connecting the I/Os of the design to it. This way, you can debug your design when you do not have access to the hardware or the hardware is remotely located.

The following ports are created:

- One 4-bit PROBE_IN0 port. This has two bits to monitor the 2-bit Sine Wave selector outputs from the finite state machine (FSM) and other two bits to mimic the state of the other two LEDs on the

board. We will configure these 4-bit signals as LEDs during run time to mimic the LEDs displayed on the KC705 board.

- One 2-bit PROBE_OUT0 port to drive the input buttons on the FSM. We will configure it so one bit can be used as a toggle switch during run time to mimic the "PUSH_BUTTON", SW3, and second bit will be used as the "PUSH_BUTTON", SW6.

Step 1: Creating a Project with the Vivado New Project Wizard

To create a project, use the New Project wizard to name the project, to add RTL source files and constraints, and to specify the target device.

1. Invoke Vivado IDE.
2. In the **Getting Started** page, click **Create New Project** to start the New Project wizard. Click **Next**.
3. In the **Project Name** page, name the new project **proj_hdl_vio** and provide the project location (`C:/Vivado_Debug`). Ensure that **Create project subdirectory** is selected. Click **Next**.
4. In the **Project Type** page, specify the **Type of Project** to create as **RTL Project**. Click **Next**.
5. In the **Add Sources** page:
   a. Set **Target Language** to **VHDL**.
   b. Click **Add Files**.
   c. In the **Add Source** Files dialog box, navigate to the `/src/Lab3 directory`.
   d. Select all VHD source files, and click **OK**.
   e. Verify that the files are added, and **Copy Sources into Project** is selected. Click **Next**.
6. In the **Add Existing IP (optional)** page:
   a. Click the **Add Files**.
   b. In the **Add Configurable IP** dialog box, navigate to the `/src/Lab3 directory`.
   c. Select all XCI source files, and click **OK**.
   d. Verify that the files are added and **Copy sources into project** is selected. Click **Next**.
7. In the **Add Constraints (optional)** dialog box, the provided XDC file, `sinegen_demo_kc705.xdc`, should automatically appear in the main window. Click **Next**.
8. In the **Default Part** page, specify the **xc7k325tffg900-2** part for the KC705 platform. You can also select **Boards** and then select **Kintex-7 KC705 Evaluation Platform**. Click **Next**.

9. Review the **New Project Summary** page. Verify that the data appears as expected, per the steps above. Click **Finish**.

   ***Note**: It might take a moment for the project to initialize.*

10. In the **Sources** window in Vivado IDE, expand sinegen_demo_inst_vio to see the source files for this lab. Note that `ila_0` core has been added to the project. However, `vio_0` (the VIO core) is missing.

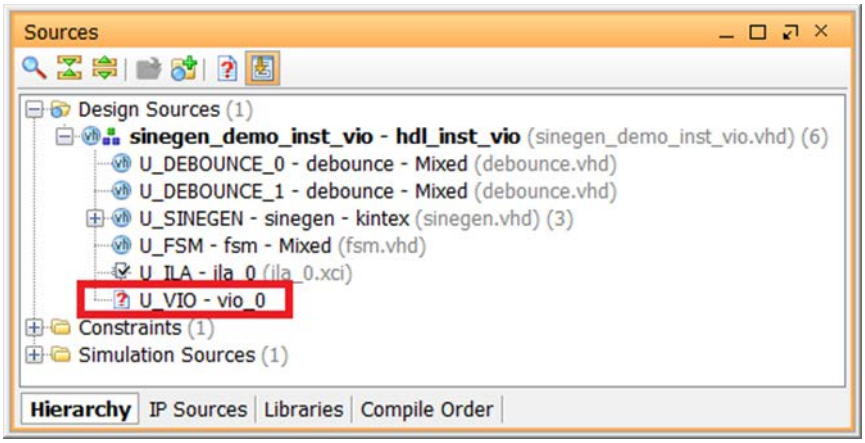


Figure 26: Missing source for VIO core

In the following step, you will instantiate and configure this VIO core.

11. From the **Flow Navigator**, click **IP Catalog**, expand **Debug & Verification**, then expand **Debug**, and double-click **VIO 3.0**. The **Customize IP** dialog box opens.

12. On the **General Options** tab, leave the **Component Name** to its default value of **vio_0**, **set Input Probe Count** to **1**, **Output Probe Count** to **1**, and select the **Enable Input Probe Activity Detectors** check box.

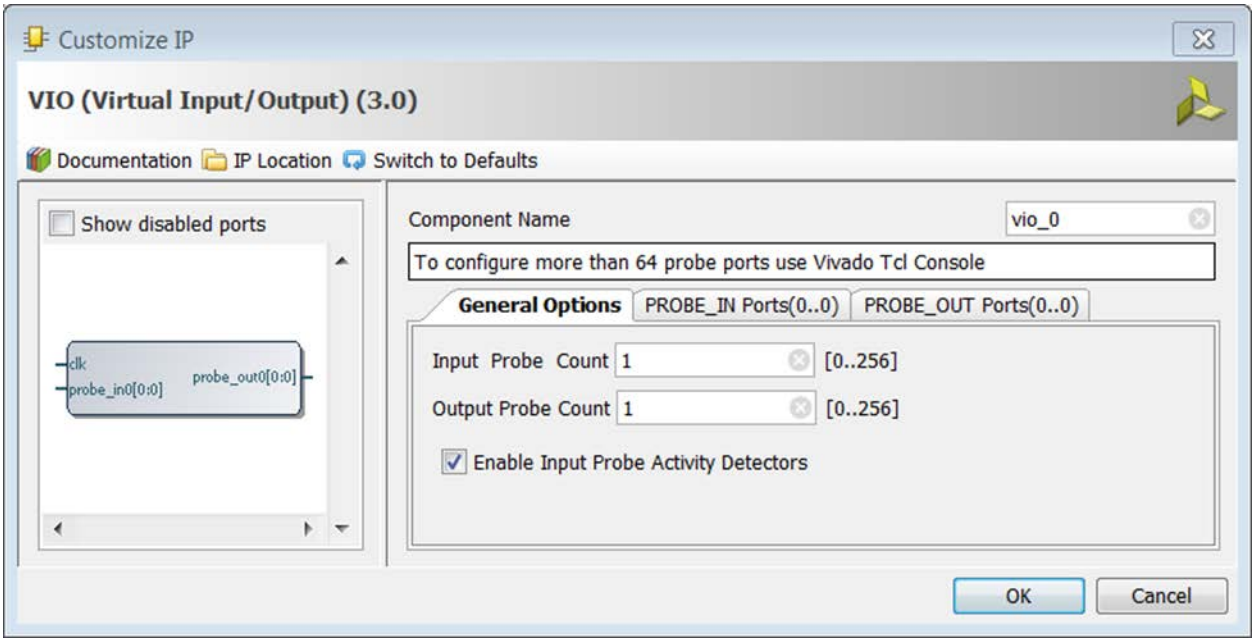
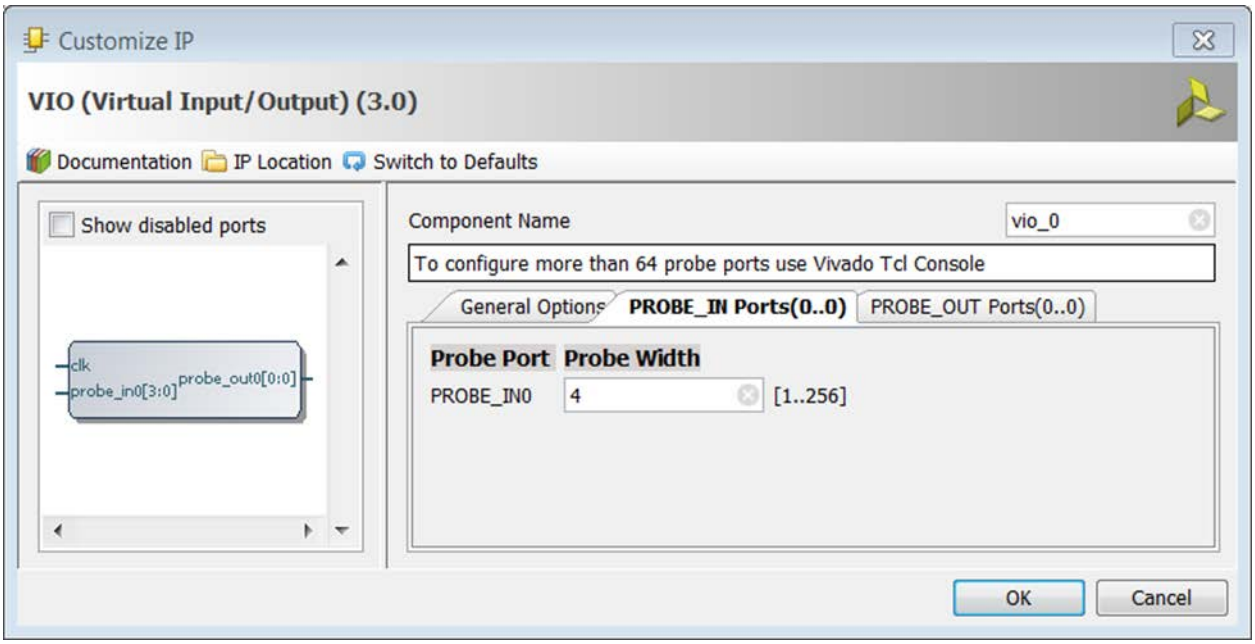


Figure 27: Configure General Options of the VIO core

13. On the **PROBE_IN Ports** tab, set **Probe Width** to **4** bits wide.



15. On **the PROBE _OUT Ports**, set **Probe Width** to **2** bits wide with an initial value of **0** in hex format.

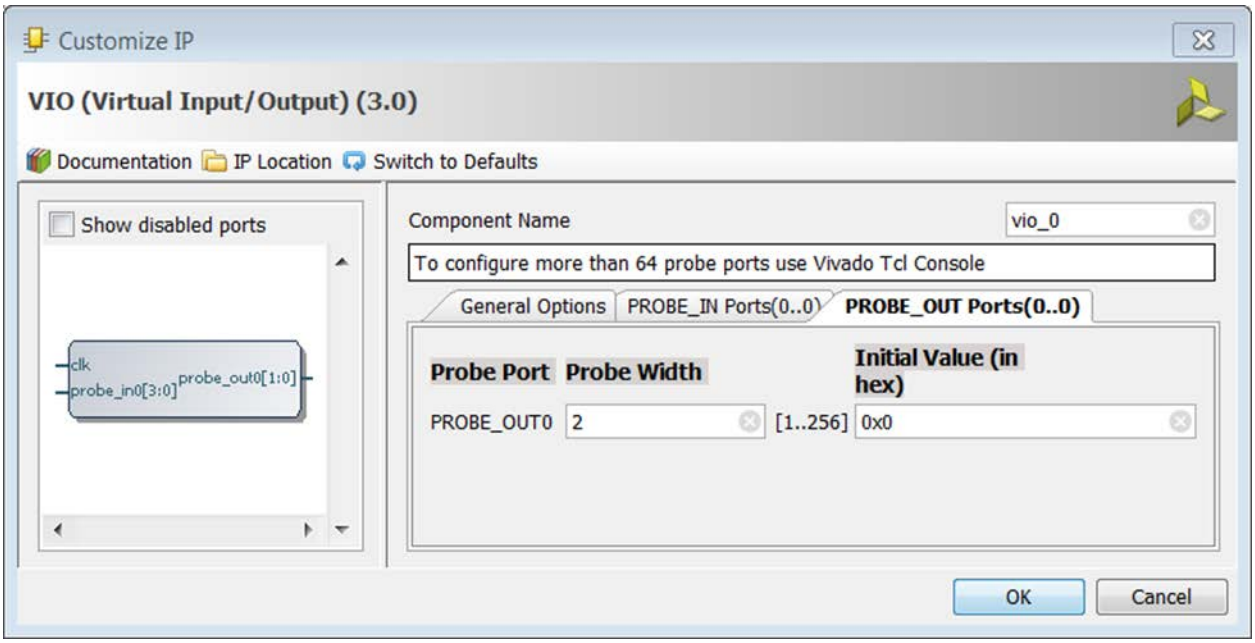


Figure 29: Configure the PROBE_OUT Ports of the VIO core

16. Click **OK** to generate the IP. The **Generate Output Products** dialog box will appear. Click **Generate**.

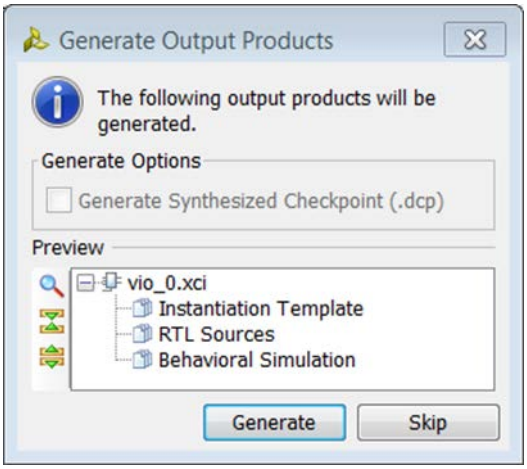


Figure 30: Generate Output Products for the VIO core

Output product generation should take less than a minute. At this point, you have finished customizing the VIO 3.0. This core has already been instantiated in the top level design as shown in the following figure.

```
------------------------------------------------------------
-- VIO
------------------------------------------------------------
U_VIO : vio_0
  port map
  (
    CLK => clk,
    PROBE_IN0 => (DONT_EAT & GPIO_BUTTONS_re(1) & sineSel),
    PROBE_OUT0(0) => push_button_reset,
    PROBE_OUT0(1) => push_button_vio
  );
```

Figure 31: VIO Instantiation in the top level design

At this point, the **Sources** window should look as shown in the following figure.

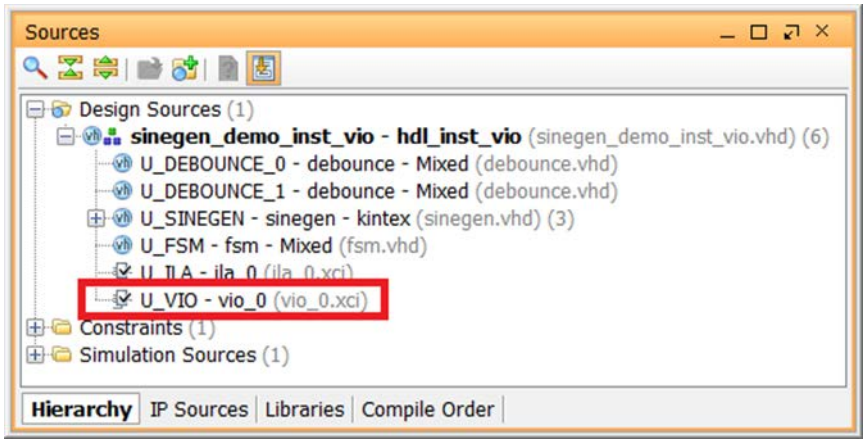


Figure 32: Instantiated VIO Core in the Sources window

17. Double-click **sinegen_demo_inst.vhd** in the **Sources** window, to open it and inspect the instantiation and port mapping of the ILA core in the HDL code. Also, note that attributes have been placed in the source file to preserve the net names.

```
71  -- Attributes for preserving the net names and to show debug nets in the synthesized netlist
72  attribute mark_debug : string;
73  attribute mark_debug of GPIO_BUTTONS_db : signal is "true";
74  attribute mark_debug of GPIO_BUTTONS_dly : signal is "true";
75  attribute mark_debug of GPIO_BUTTONS_re : signal is "true";
76  attribute mark_debug of sineSel : signal is "true";
77  attribute mark_debug of sine : signal is "true";
78  attribute mark_debug of push_button_vio : signal is "true";
79  attribute mark_debug of push_button_reset : signal is "true";
80  attribute mark_debug of DONT_EAT : signal is "true";
```

Figure 33: Using "mark_debug" attribute to preserve net names and show debug nets in the synthesized netlist

```
------------------------------------------------------------------
-- ILA
------------------------------------------------------------------
U_ILA : ila_0
  port map
  (
    CLK => clk,
    PROBE0 => sineSel,
    PROBE1 => sine,
    PROBE2 => (push_button_reset & push_button_vio),
    PROBE3 => GPIO_BUTTONS_re,
    PROBE4 => GPIO_BUTTONS_dly
  );
```

Figure 34: Hook signals that need to be debugged in the ILA

Step 2: Synthesize, Implement, and Generate Bitstream

1. From the **Program and Debug** drop-down list, in Flow Navigator, click **Generate Bitstream**. This synthesizes, implements, and generates a bitstream for the design.
2. **The No Implementation Results Available** dialog box appears. Click **Yes**.
3. After bitstream generation completes, the **Bitstream Generation Completed** dialog box appears. **Open Implemented Design** is selected by default. Click **OK**.

4. From the **Implementation** drop-down list in Flow Navigator, expand **Implemented Design** and select **Report Timing Summary**.

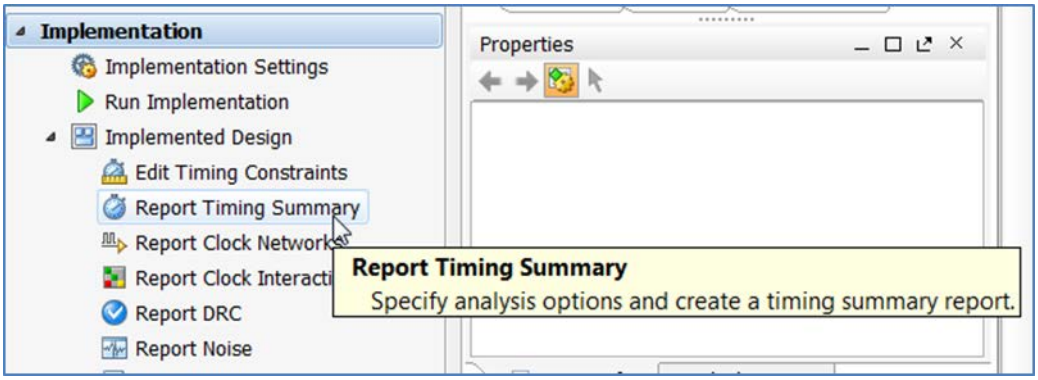
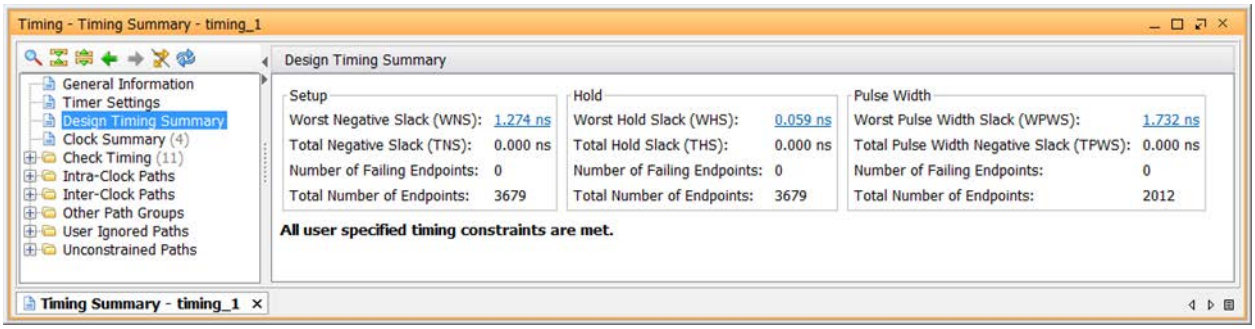


Figure 35: Report timing summary for Lab 3

5. In the **Report Timing Summary** dialog box, click **OK**. Make sure that all timing constraints have been met.


Timing - Timing Summary - timing_1
General Information
Timer Settings
Design Timing Summary
Clock Summary (4)
Check Timing (11)
Intra-Clock Paths
Inter-Clock Paths
Other Path Groups
User Ignored Paths
Unconstrained Paths
Design Timing Summary
Setup
Worst Negative Slack (WNS): 1.274 ns
Total Negative Slack (TNS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 3679
Hold
Worst Hold Slack (WHS): 0.059 ns
Total Hold Slack (THS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 3679
Pulse Width
Worst Pulse Width Slack (WPWS): 1.732 ns
Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 2012
All user specified timing constraints are met.
Timing Summary - timing_1

6. Proceed to Using Vivado® Logic Analyzer to Debug Hardware. Skip forward to **Verifying the VIO Core Activity (Only applicable to Lab 3)** to complete the rest of this lab.

Lab 4: Using Synplify Pro Synthesis Tool and Vivado for Debugging a Design

Introduction

This simple tutorial shows how to do the following:

- Create a Synplify Pro project for the wave generator design.
- Mark nets for debug in the Synplify Pro constraints file as well as VHDL source files.
- Synthesize the Synplify Pro project to create an EDIF netlist.
- Create a Vivado® project based on the Synplify Pro netlist.
- Use the Vivado IDE to setup and debug the design from the synthesized design using Synplify Pro (Version 2013-3 SP1).

Step 1: Create a Synplify Pro Project

1. Launch Synplify Pro and select **File > New**. Set **File Type** to **Project File (Project)** as highlighted in the following figure. In the **New File Name** box, enter **synplify_1**. Click **OK**.

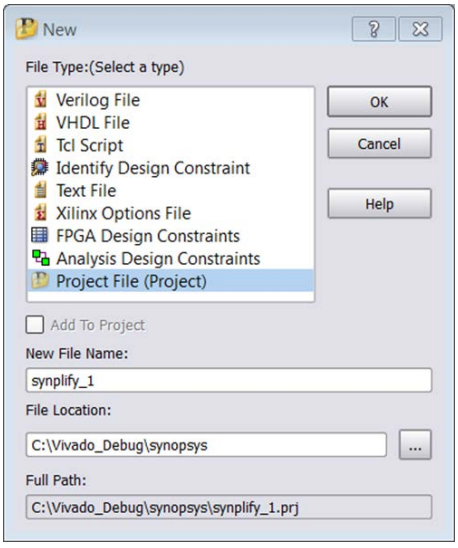


Figure 37: Synplify Pro New Project Dialog Box

2. If you get a dialog box asking you to create a non-existing directory click **OK**.

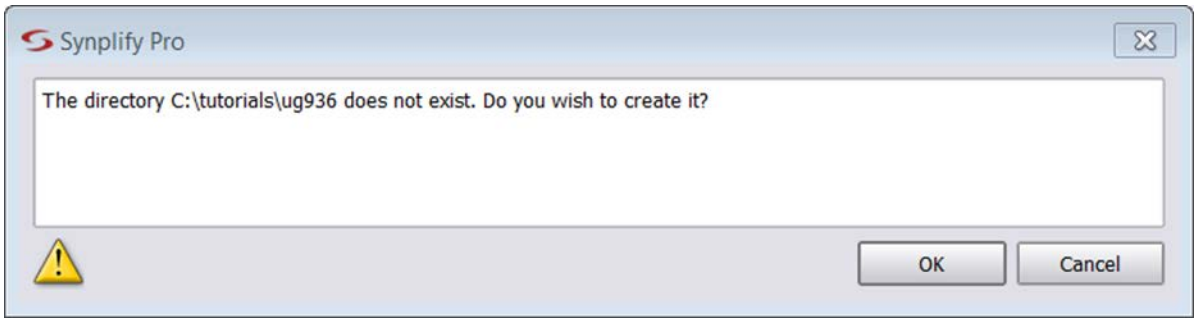


Figure 38: Synplify Pro project confirmation dialog box

3. In the left panel of the **Synplify Pro** window, click **Add File** as shown in the following figure.

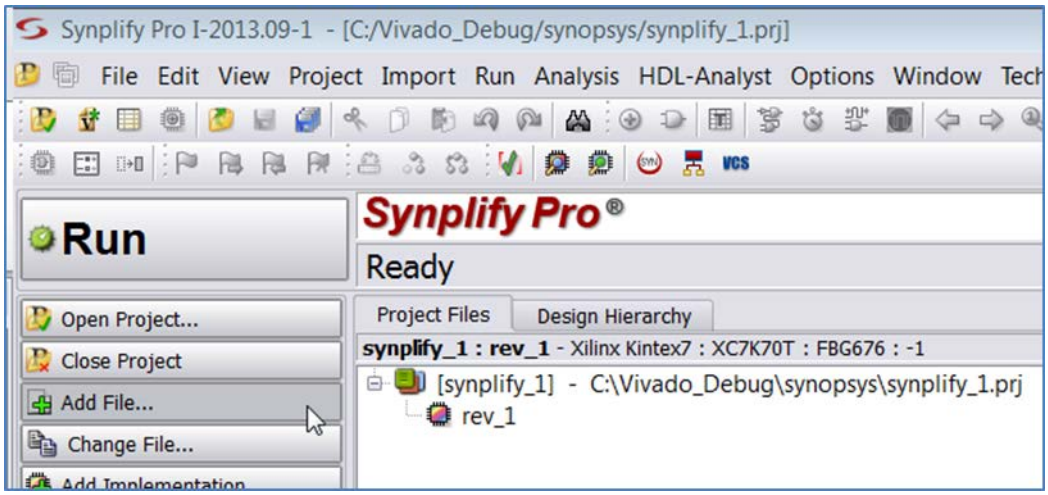


Figure 39: Adding Files to a Synplify Pro Project

4. In the **Add Files to Project** dialog box, change the **Files of Type** to **HDL File**. Navigate to `C:\Vivado_Debug\src\Lab4`, which shows all the VHDL source files needed for this lab. Select the following three files by pressing the **Ctrl** key and clicking on them.

- `debounce.vhd`
- `fsm.vhd`
- `sinegen_demo.vhd`

Click **Add**.

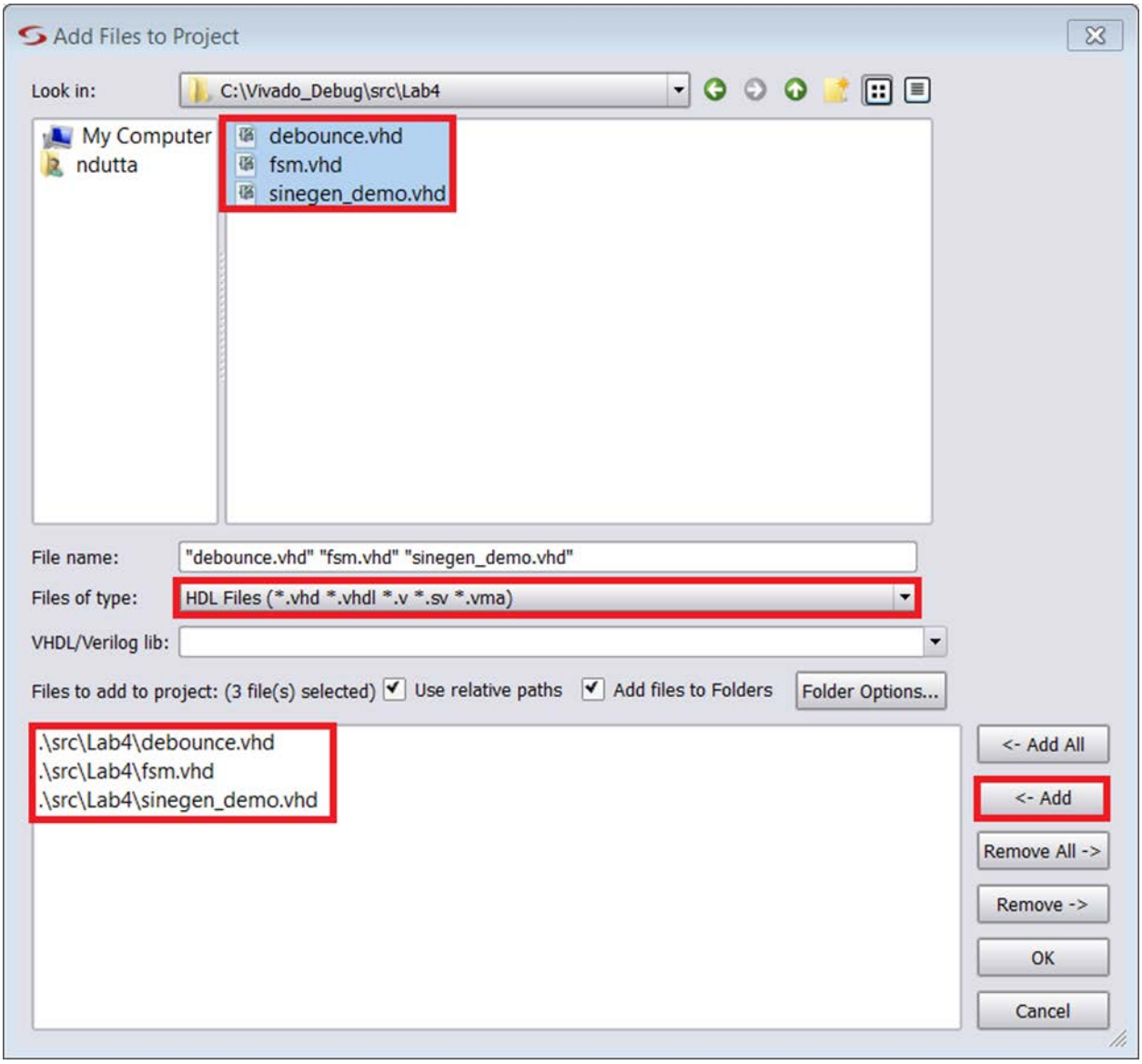


Figure 40: Adding VHDL Source Files to the Synplify Pro Project

6. In the same dialog box set **Files of type** to **Constraints File**. This shows the `synplify_1.sdc` file. Select the file and click **Add** as shown in the following figure.

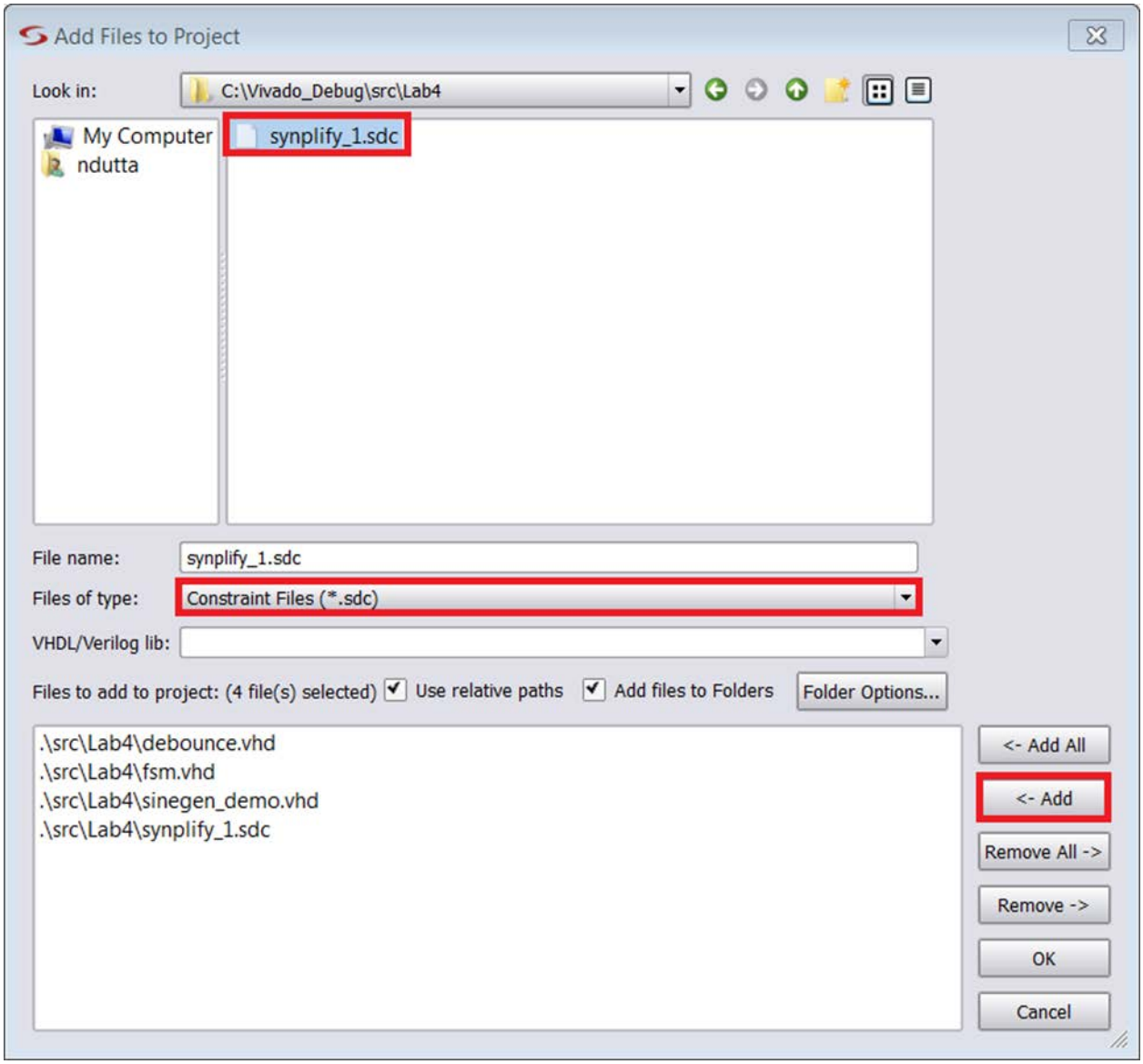


Figure 41: Adding SDC Constraints File to the Synplify Pro Project

7. In the same dialog box set **Files of type to Compiler Directives File**. This shows the `synplify_1.cdc` file. Select the file and click **Add** as shown in the following figure. Click **OK**.

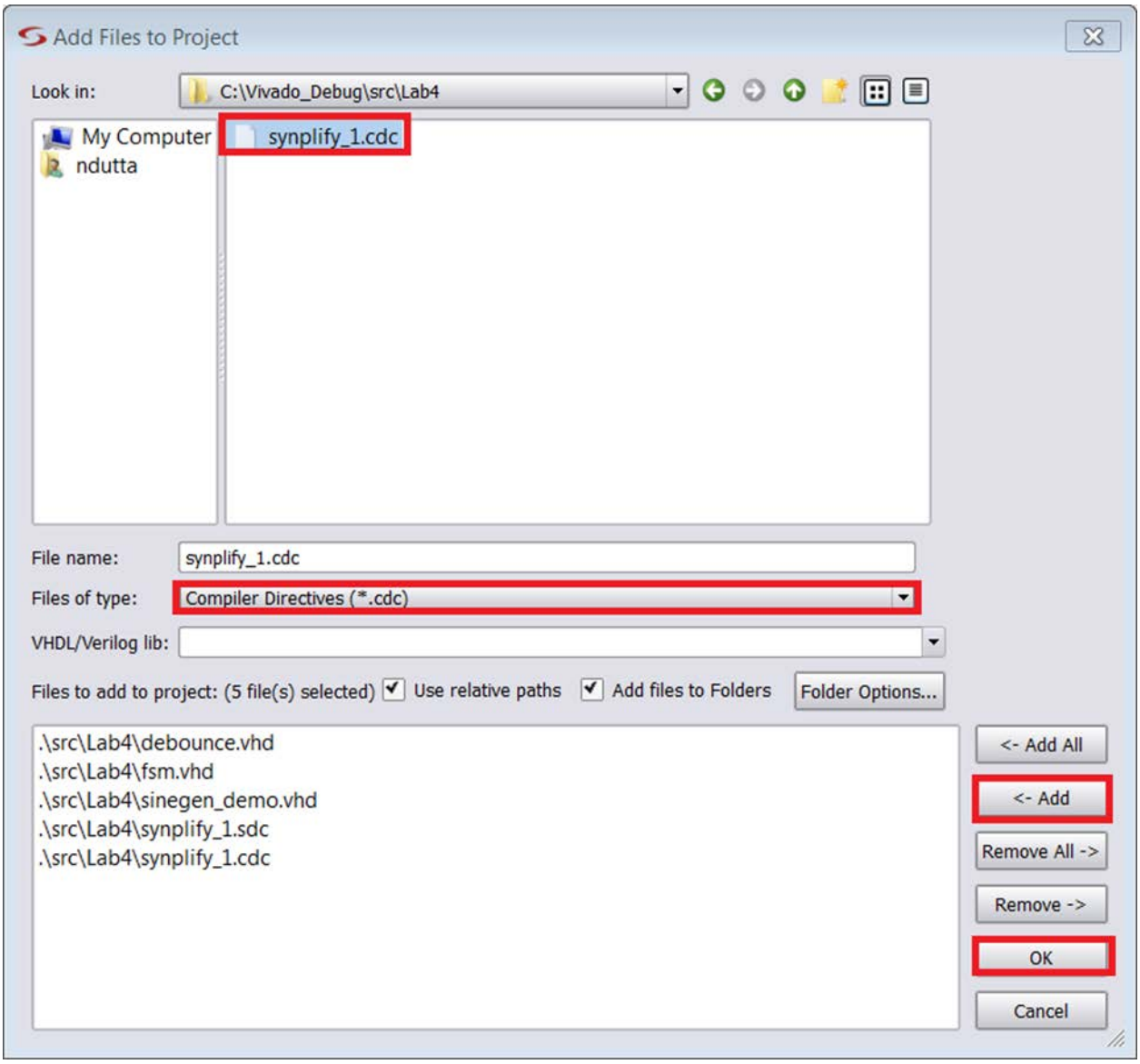


Figure 42: Adding CDC Constraints File to the Synplify Pro Project

8. Now, you need to set the implementation options. Click **Implementation Options** in the **Synplify Pro** window as shown in the following figure.

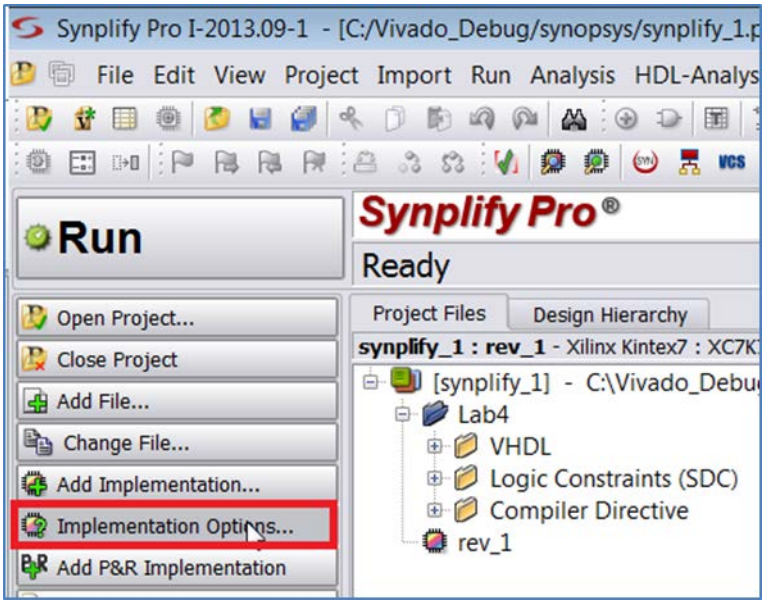


Figure 43: Opening Implementation Options in Synplify Pro

9. This brings up the **Implementation Options** dialog box as shown in the following figure. In the **Device** tab, set **Technology** to **Xilinx Kintex7**, **Part** to **XC7K325T**, **Package** to **FFG900** and **Speed** to **-2**. Leave all the other options at their default values. Click **OK**.

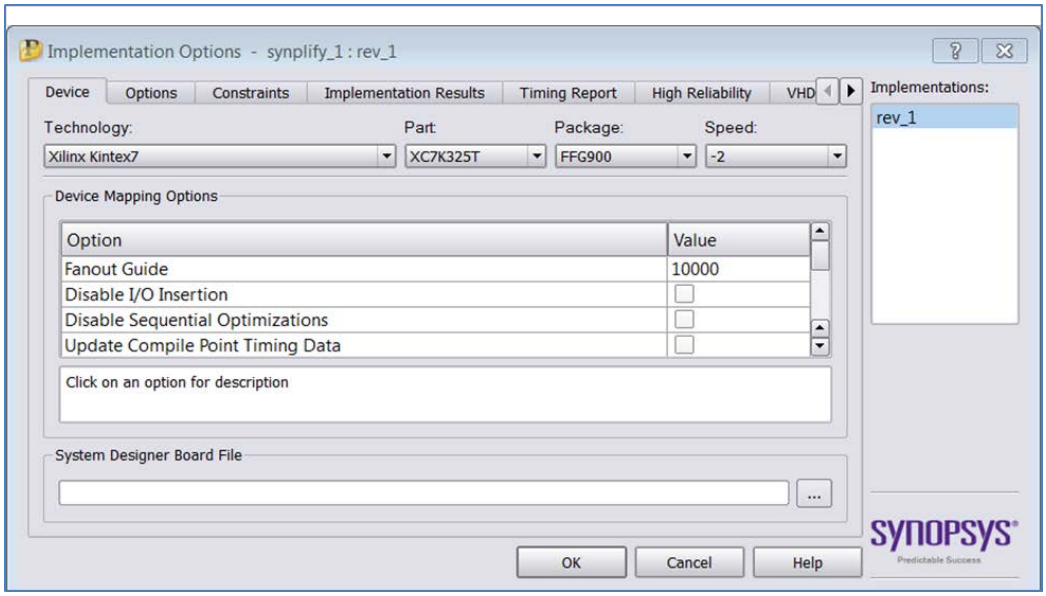


Figure 44: Specifying Implementation Options in Synplify Pro

10. You need to preserve the net names that you want to debug by putting attributes in the HDL files. These attributes are already placed in the `sinegen_demo.vhd`, file of this tutorial. Open the `sinegen_demo.vhd` file and inspect the lines shown.

```
-- Attributes for Synplify Pro
attribute syn_keep : boolean;
attribute syn_keep of GPIO_BUTTONS_db   : signal is true;
attribute syn_keep of GPIO_BUTTONS_dly  : signal is true;
attribute syn_keep of GPIO_BUTTONS_re   : signal is true;
```

Figure 45: Specifying attributes to preserve net names in Synplify

11. You also can specify the mark_debug attributes in the source HDL files to mark the signals for debug, as shown in the snippet code from `singen_demo.vhd` file.

```
-- Add mark_debug attributes to show debug nets in the synthesized netlist
attribute mark_debug : string;
attribute mark_debug of GPIO_BUTTONS_db : signal is "true";
attribute mark_debug of GPIO_BUTTONS_dly : signal is "true";
attribute mark_debug of GPIO_BUTTONS_re : signal is "true";
```

Figure 46: Add mark_debug attribute in HDL file

12. The `synplify_1.sdc` file contains various kinds of constraints such as pin location, I/O standard, and clock definition. The `synplify_1.cdc` file contains directives for the compiler. Here is where the nets of interest to us that are marked for debug are located. The attribute and the nets selected for debug are shown in the following figure.

```
# Attributes that are needed to mark_debug the nets that are needed to be viewed in ILA

define_attribute  -comment {Mark sinegen as black box} {v:work.sinegen} {syn_black_box} {1}
define_attribute  -comment {Set no_prune on sinegen} {v:work.sinegen} {syn_noprune} {1}
define_attribute  -comment {Mark entire bus for debug} {i:sinegen.sine[*]} {mark_debug} {"true"}
define_attribute  -comment {Mark entire bus for debug} {i:sinegen.sel[*]} {mark_debug} {"true"}
~
```

Figure 47: Synplify Pro Constraints in CDC Files

In the above constraints, sinegen has been defined as a black box by using the syn_black_box attribute. Second, the syn_no_prune attribute has been used so that the I/Os of this block are not optimized away. Finally, two nets, `sine[20:0]` and `sel[1:0]` have been assigned the

mark_debug attribute such that these two nets should show up in the synthesized design in Vivado IDE for further debugging. For further information on these attributes, please refer to the *Synplify Pro User Manual* and *Synplify Pro Reference Manual*.

Step 2: Synthesize the Synplify Project

1. Before implementing the project, you need to set the name for the output netlist file. By default, the name of the output netlist file is synplify_1.edf. To change the name of the output file, type the following command at the Tcl command prompt:

   ```
   %project -result_file "./rev_1/sinegen_demo.edf"
   ```

   You will use this file in Vivado IDE.

2. With all the project settings in place, click the **Run** button in the left panel of the **Synplify Pro** window to start synthesizing the design.

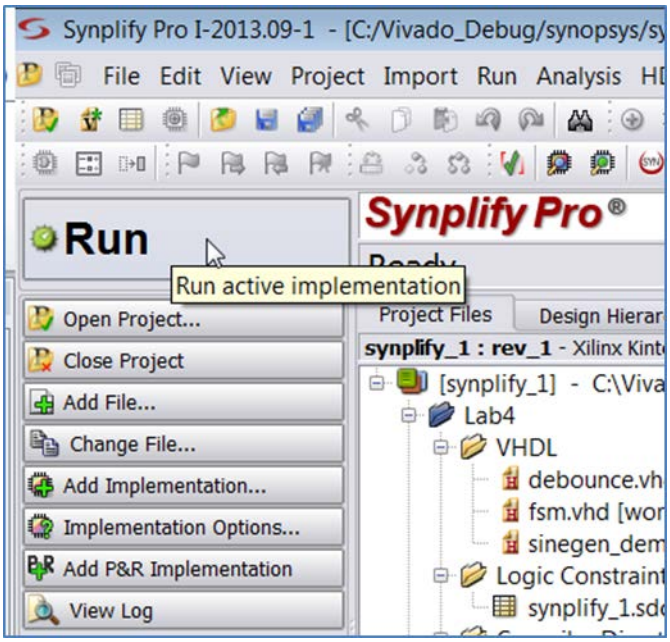


Figure 48: Synthesize the design in Synplify

3. During synthesis, status messages appear in the **Tcl Script** tab. Warning messages are expected, but there should not be any Error messages. To see detailed messages, click the **Messages** tab in the bottom left-hand corner of the Synplify Pro console.

4. When synthesis completes, the output netlist is written to the file:

   `rev_1/sinegen_demo.edf.`

   [Optional] To view the netlist select **View > View Result File**.

5. Click **File > Save All** to save the project, then click **File > Exit**.

Step 3: Create EDIF Netlists for the Black Box Created in Synplify Pro

The black box, sinegen, created in the Synplify Pro project, contains the Direct Digital Synthesizer IP. You need to create a synthesized design for this block. To do this, create an RTL type project in Vivado IDE by following the steps outlined below.

1. Launch Vivado IDE.
2. Click **Create New Project**. This opens up the **New Project** wizard. Click **Next**.
3. Under **Project Name**, set the project name to **proj_synplify_netlist**. Click **Next**.
4. Under **Project Type**, select **RTL Project**. Click **Next**.
5. Under **Add Sources**, click **Add Files**, navigate to the `Vivado_Debug/src/Lab4` folder and select the `sinegen.vhd` file. Set **Target Language** to **VHDL**. Ensure that **Copy sources into project box** is selected. Click **Next**.
6. Under **Add Existing IP**, click **Add Files**, navigate to the `Vivado_Debug/src/Lab4` folder and select the `sine_high.xci`, `sine_low.xci`, and `sine_mid.xci` files. Click **Next**.
7. Under **Add Constraints**, the `.sdc` files are automatically added to the project. These files are not needed for this step. Remove them from this project by clicking **Remove Selected File** on the right of the dialog box. Click **Next**.
8. Under **Default Part**, select **Boards** and then select the **Kintex-7 KC705 Evaluation Platform** and correct version for your hardware. Click **Next**.
9. Under **New Project Summary**, ensure that all the settings are correct. Click **Finish**.
10. Once the project has been created, in Vivado Flow Navigator, under the Project Manager folder, click **Project Settings**. In the pop-up dialog box, in the left panel, click **Synthesis**. From the pull down menu on the right panel, set **-flatten_hierarchy** to **none**. Click **OK**.
11. In Vivado IDE Flow Navigator, under **Synthesis Folder**, click **Run Synthesis**.
12. When synthesis completes the **Synthesis Completed** dialog box appears. Select **Open Synthesized Design** and click **OK**.

13. Now you need to write the netlist file for all the components used in the sinegen block. The four netlist files used in this tutorial are already provided as a part of the source files. However, you can overwrite them by using your own netlist files. To do this use the following Tcl command in the Tcl console of Vivado IDE.

    ```
    write_edif  -force ../Vivado_Debug/src/Lab4/sinegen.edn
    ```

    Ensure that the path specified to the `src` folder is correct. At this point, you should see four `.edn` files in the `Vivado_Debug/src` folder as shown below:

    - `dds_compiler_v6_0_viv.edn`
    - `dds_compiler_v6_0_viv_parameterized1.edn`
    - `dds_compiler_v6_0_viv_parameterized3.edn`
    - `sinegen.edn`

14. Click **File > Exit** in Vivado IDE. When the **OK to exit** dialog box pops up, click **OK**.

Step 4: Create a Post Synthesis Project in Vivado IDE

1. Launch Vivado IDE.
2. Click **Create New Project**. This opens up the New Project wizard. Click **Next**.
3. Set the **Project Name** to **proj_synplify**. Click **Next**.
4. Under **Project Type**, select **Post-synthesis Project**. Click **Next**.
5. Under **Add Netlist Sources**, click **Add Files**, navigate to the `Vivado_Debug/synopsys/rev_1` folder, and select **sinegen_demo.edf**. Click **OK**.
6. Add the four netlist files created in the previous section. Click **Add Files** again, navigate to the `Vivado_Debug/src/Lab4` folder and select the following files:

- `sinegen.edn`
- `dds_compiler_v6_0_viv.edn`
- `dds_compiler_v6_0_viv_parameterized1.edn`
- `dds_compiler_v6_0_viv_parameterized3.edn`

  Click **OK** in the **Add Source Files** dialog box. In the **Add Netlist Sources** dialog box ensure that **Copy Sources into Project** is selected. Click **Next**.

7. Under **Add Constraints**, a `.sdc` file. should be automatically populated. Remove this file by selecting it and clicking **Remove Selected File** on the right of the dialog box. Click **Add Files** , navigate to the `Vivado_debug/src` folder, and select the **sinegen_demo_kc705.xdc** file. This file has the appropriate constraints needed for this Vivado project. Click **OK** in the **Add Constraints File**

dialog box. In the **Add Constraints (optional)** dialog box ensure that **Copy Constraints into Project** is selected. Click **Next**.

8. Under **Default Part**, select **Boards** and then select **Kintex-7 KC705 Evaluation Platform** and the right version number for your hardware. Click **Next**.
9. Under **New Project Summary**, ensure that all the settings are correct and click **Finish**.

10. In the **Sources** window, select **sinegen_demo.edf** and select **Specify Top Module**.

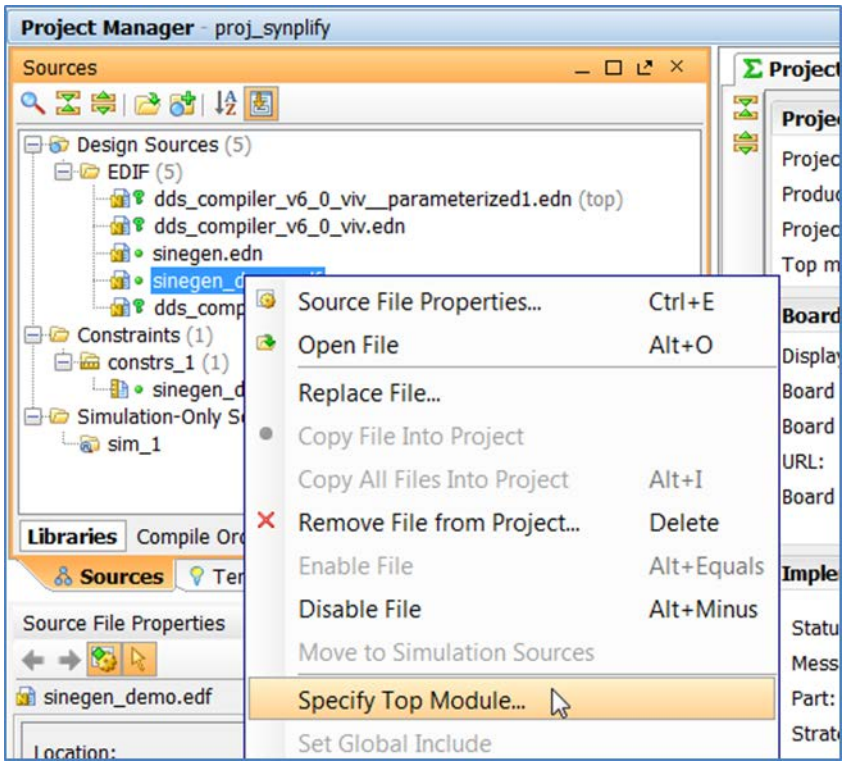


Figure 49: Specifying the top-level module

11. In the **Specify Top Module** dialog box, click **Browse**.

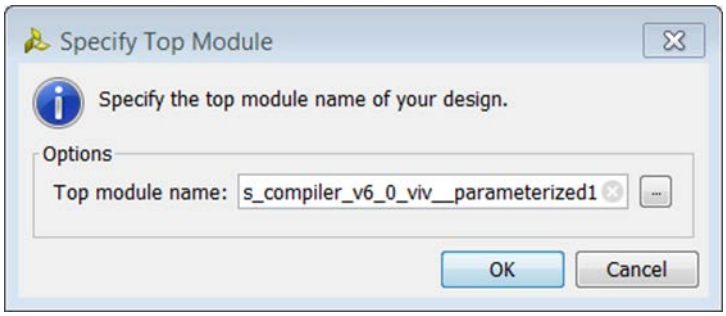


Figure 50: Browse to the top module

12. In the **Select Top Module** dialog box, select **sinegen_demo**, then click **OK**.

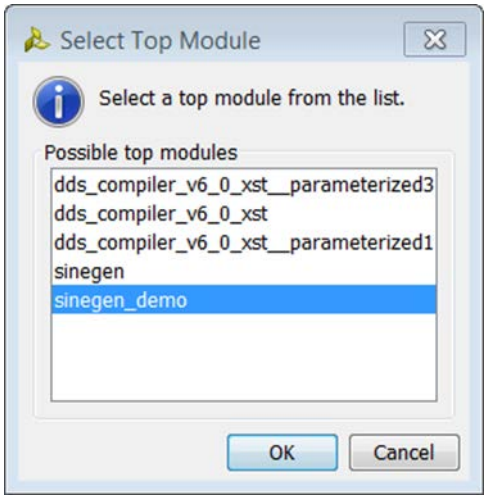


Figure 51: Select the top level module

13. Click **OK** in the **Specify Top Module** dialog box after ensuring that the top level module is correct.

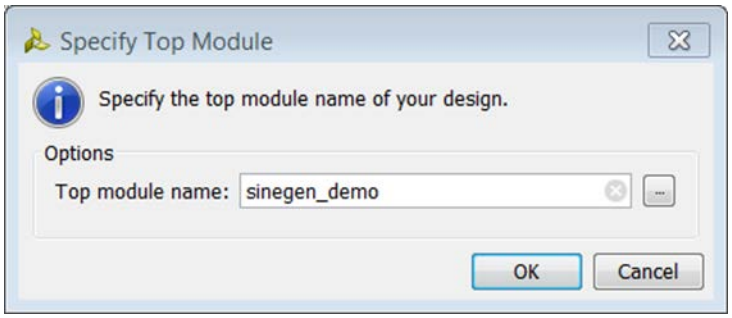


Figure 52: Specify sinegen_demo as the top level module

Step 5: Add (more) Debug Nets to the Project

1. In Vivado IDE, in the Flow Navigator, select **Open Synthesized Design** from the **Netlist Analysis** folder.
2. Select the **Netlist** tab in the **Netlist** window to expand Nets. Select the following nets for debugging:

- `GPIO_BUTTONS_c(2)`
- `sine (20)`
- `sineSel (2)`

After selecting all the nets mentioned above click **Mark Debug**.

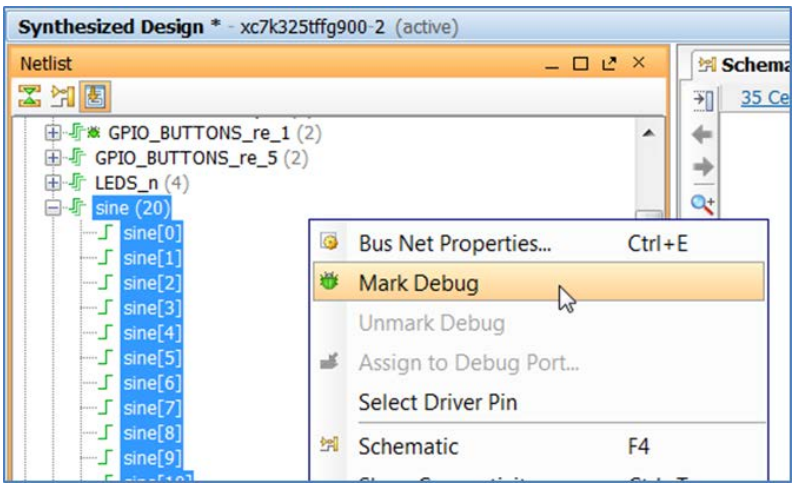


Figure 53: Mark additional signals for debug

3. In the **Confirm Debug Net(s)** dialog box, click **OK**.
4. You should be able to see all the nets that are marked for debug as shown in the following figure.

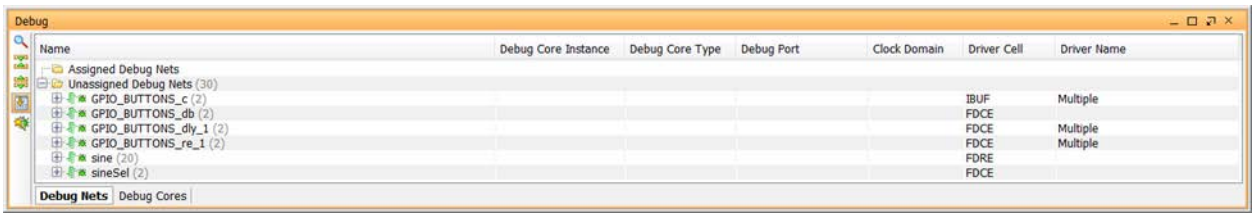


Figure 54: Nets added for debug through the Synplify Pro Flow in Vivado IDE

Running the Set up Debug Wizard

5. Click the **Set up Debug** icon in the **Debug** window or select the **Tools** menu, and select **Set up Debug**. The **Set up Debug** wizard opens.

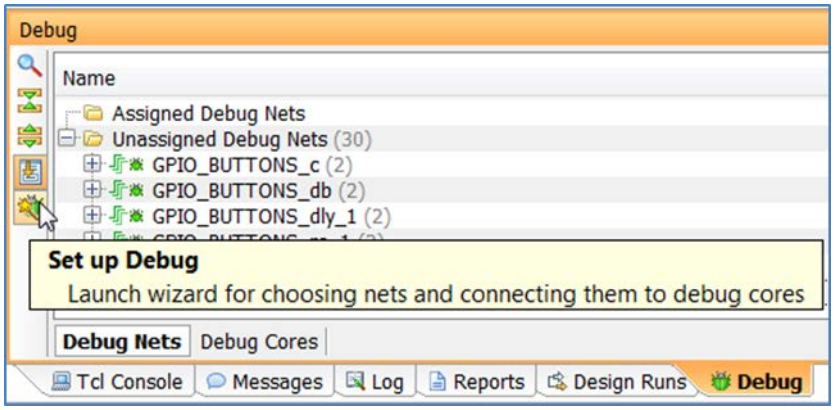


Figure 55: Run the Set up Debug wizard

6. Click through the wizard to create Vivado logic analyzer debug cores, keeping the default settings.

   ***Note**: In the **Specify Nets to Debug** dialog box, ensure that all the nets marked for debug have the same clock domain.*

Step 6: Implementing the Design and Generating the Bitstream

1. In the Flow Navigator, under the **Program and Debug** drop-down list, click **Generate Bitstream**.
2. In the **Save Project** dialog box, click **Save**.
3. When the Bitstream generation finishes, the **Bitstream Generation Completed** dialog box pops-up and **Open Implemented Design** is selected by default. Click **OK**.
4. If you get a dialog box asking to close the synthesized design before opening the implemented design, click **Yes**.
5. Proceed to Using Vivado® Logic Analyzer to Debug Hardware to complete the rest of this lab.

Using Vivado Logic Analyzer to Debug Hardware

Introduction

The final step in debugging is to connect to the hardware and debug your design using the Integrated Logic Analyzer. Before continuing, make sure you have the KC705 hardware plugged into a machine.

In this step, you learn:

- How to debug the design using the Vivado® logic analyzer.
- How to use the currently supported Tcl commands to communicate with your target board (KC705).
- How to discover and correct a circuit problem by identifying unintended behaviors of the push button switch.
- Some useful techniques for triggering and capturing design data.

Step 1: Verifying Operation of the Sine Wave Generator

After doing some setup work, you will use Vivado logic analyzer to verify that the sine wave generator is working correctly. Your two primary objectives are to verify that:

- All sine wave selections are correct.
- The selection logic works correctly.

Setting Up

Connecting to the target board remotely

If you plan to connect remotely, you will need to make sure you have KC705 hardware plugged into a machine and you are running a cse_server application on that machine. If you plan to connect locally, skip steps 1-3 below.

1. Connect the Digilent USB JTAG cable of your KC705 board to a USB port on a Windows system.
2. Ensure that the board is plugged in and powered on.
3. Assuming you are connecting your KC705 board to a 64-bit Windows machine and you will be running the hw_server from the network instead of your local drive, open a `cmd` prompt and type the following:

   ```
   <Xilinx_Install>\Vivado\2014.1\bin\hw_server
   ```

Leave this cmd prompt open while the hw_server is running. Note the machine name that you are using, you will use this later when opening a connection to this instance of the hw_server application.

Connecting to the target board locally

If you plan to connect locally, ensure that you have your KC705 hardware plugged into a Windows machine and then perform the following steps:

1. Connect the Digilent USB JTAG cable of your KC705 board to a USB port on a Windows system.
2. Ensure that the board is plugged in and powered on.
3. Turn DIP switch positions (pin 1 on SW13, De-bounce Enable) to the OFF position.

Using the Vivado Integrated Logic Analyzer

1. In the Flow Navigator, from the **Program and Debug** drop-down list, select **Open Hardware Manager**.

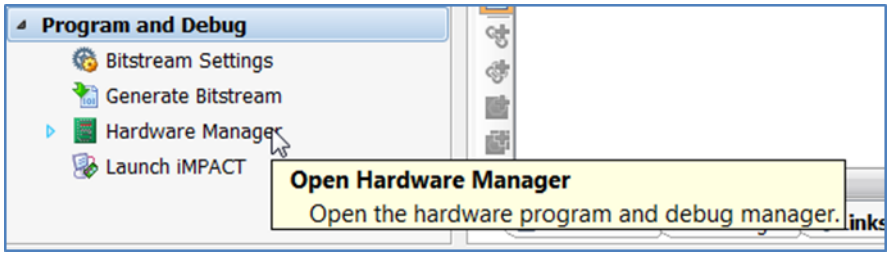


Figure 56: Open Hardware Manager

2. Click **Open a new hardware target** in the **Hardware Session** view.

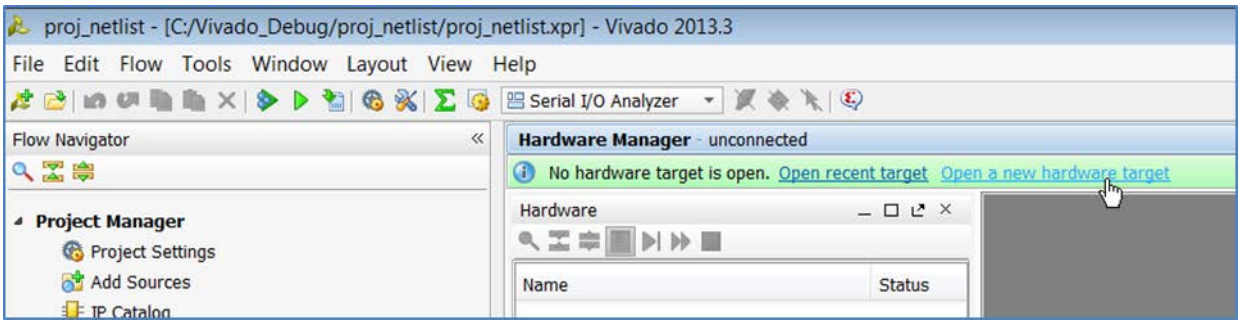


Figure 57: Connect to a Hardware Target

3. The **Hardware Manager** window opens. Click **Open a new hardware target**. The **Open New Hardware Target** wizard opens.

4. In the **Hardware Server Settings** page, type the name of the server (or select **Local server** if the target is on the local machine) in the **Connect to** field.

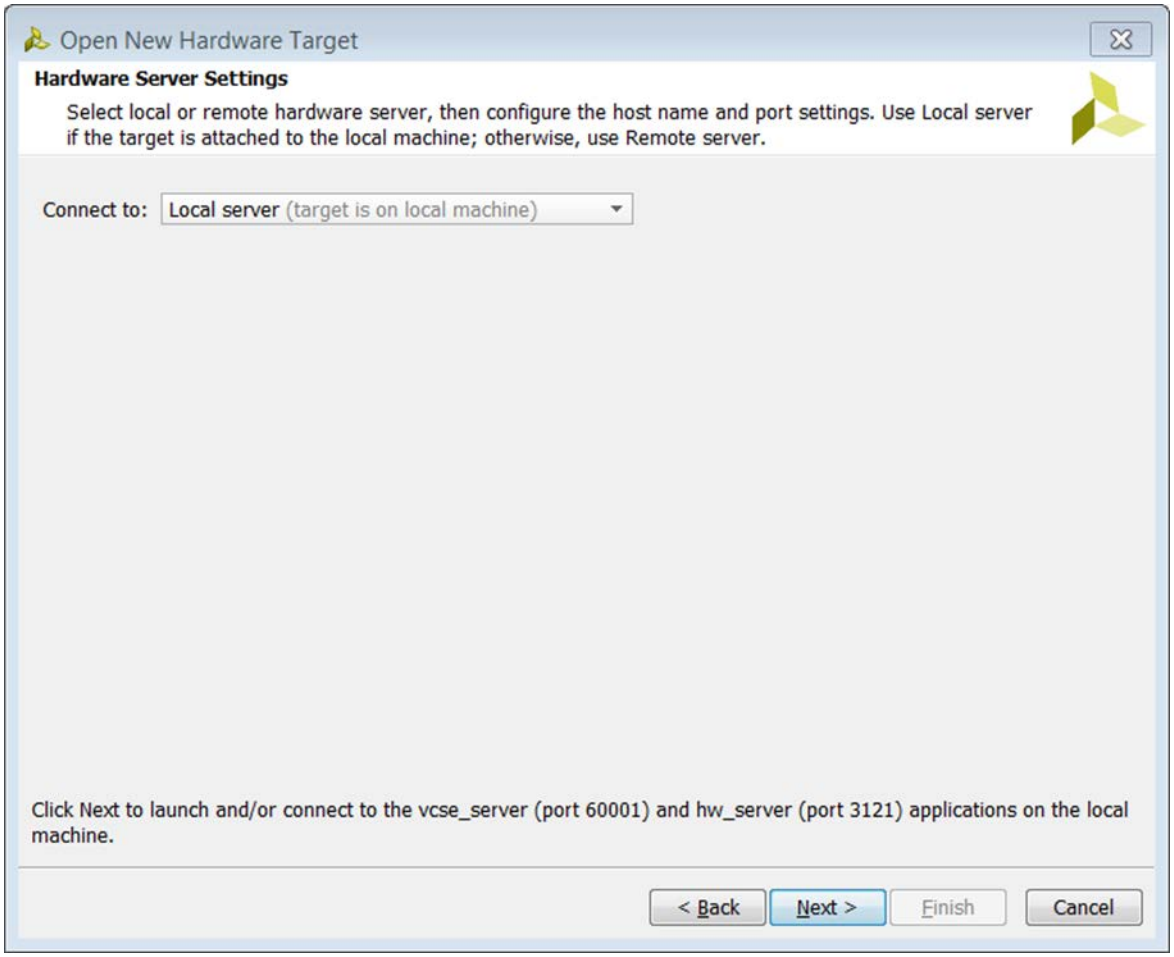

Open New Hardware Target
Hardware Server Settings
Select local or remote hardware server, then configure the host name and port settings. Use Local server if the target is attached to the local machine; otherwise, use Remote server.
Connect to:
Local server (target is on local machine)
Click Next to launch and/or connect to the vcse_server (port 60001) and hw_server (port 3121) applications on the local machine.
< Back
Next >
Finish
Cancel

Figure 58: Hardware Server Settings

***Note**: Depending on your connection speed, this may take about 10 to 15 seconds.*

5. If there is more than one target connected, you will see multiple entries in the **Select Hardware Target** page. In this tutorial, there is only one target, as shown in the following figure. Click **Next**.

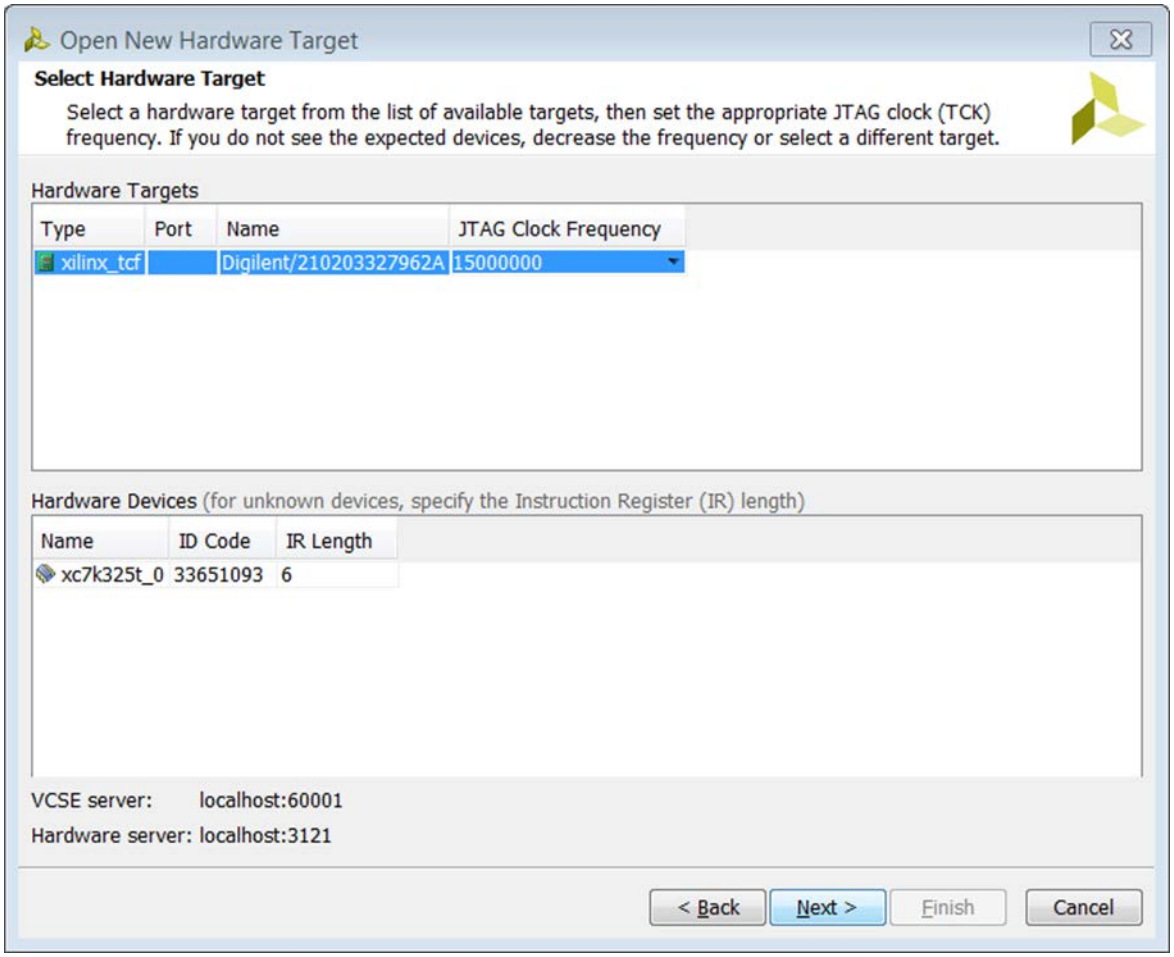


Figure 59: Select Hardware Target

6. Leave these settings at their default values. Click **Next**.

7. In the **Open Hardware Target Summary** page, click **Finish** as shown in the following figure.

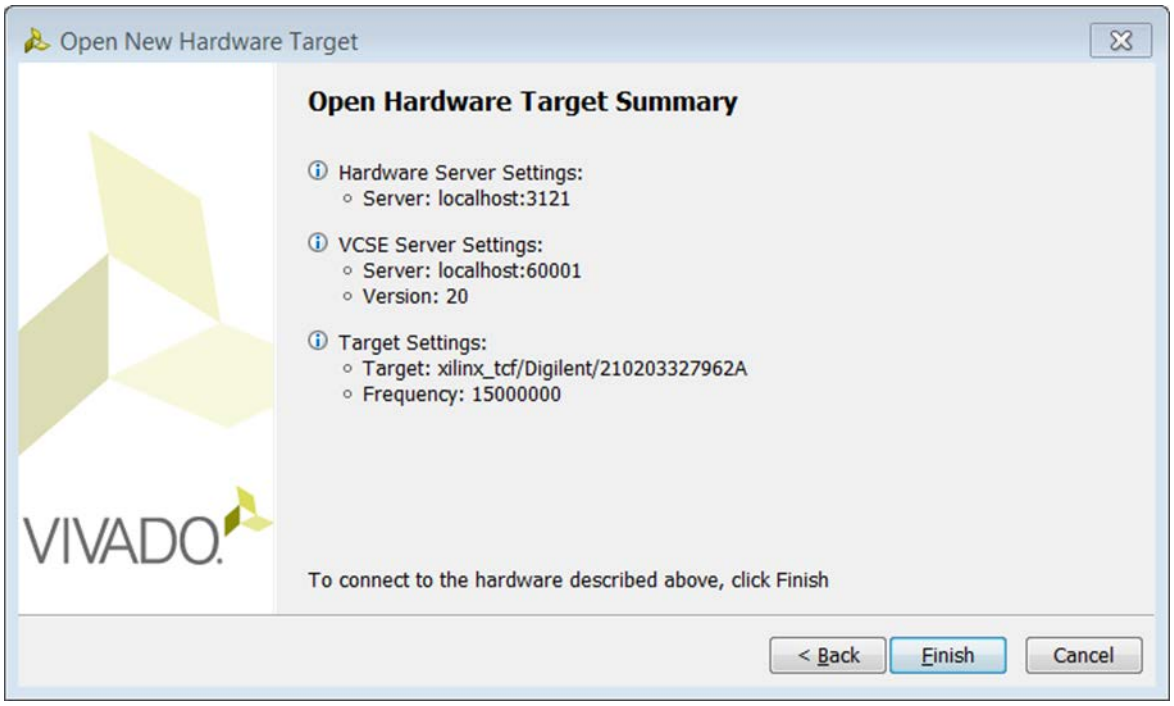


Figure 60: Hardware Target Summary

8. Wait for the connection to the hardware to complete. The dialog in following figure appears while hardware is connecting.

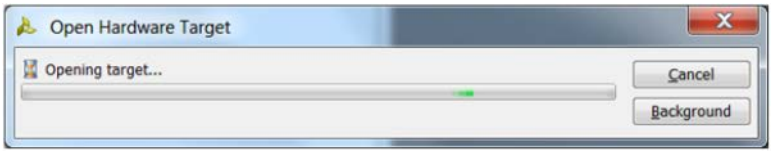


Figure 62: Open Hardware Target

After the connection to the hardware target is made, the dialog shown in the following figure appears.

***Note**: The **Hardware** tab in the **Debug** view shows the hardware target and XC7K325T device detected in the JTAG chain.*

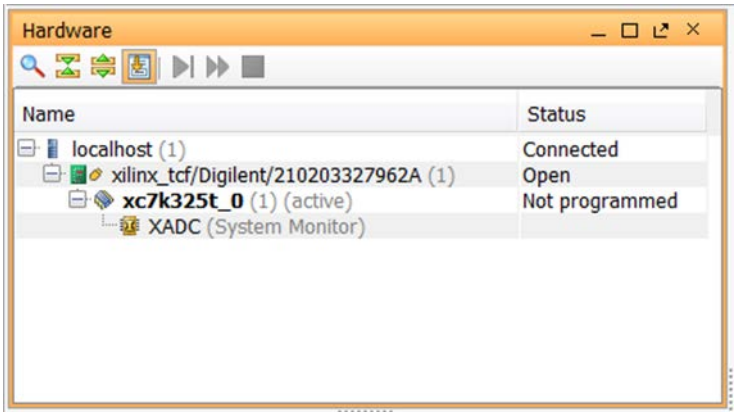


Figure 63: Active Target Hardware

9. Next, program the XC7K325T device using the `.bit` bitstream file that was created previously by right-clicking the **XC7K325T** device and **selecting Program Device** as shown in the following figure.

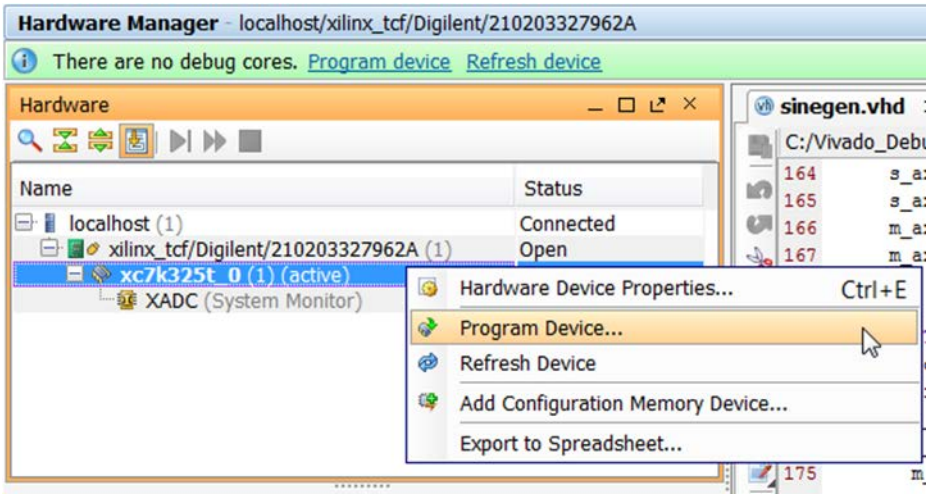


Figure 64: Program Active Target Hardware

10. In the **Program Device** dialog box verify that the `.bit` file is correct for the lab that you are working on and click **Program** to program the device as shown in the following figure.

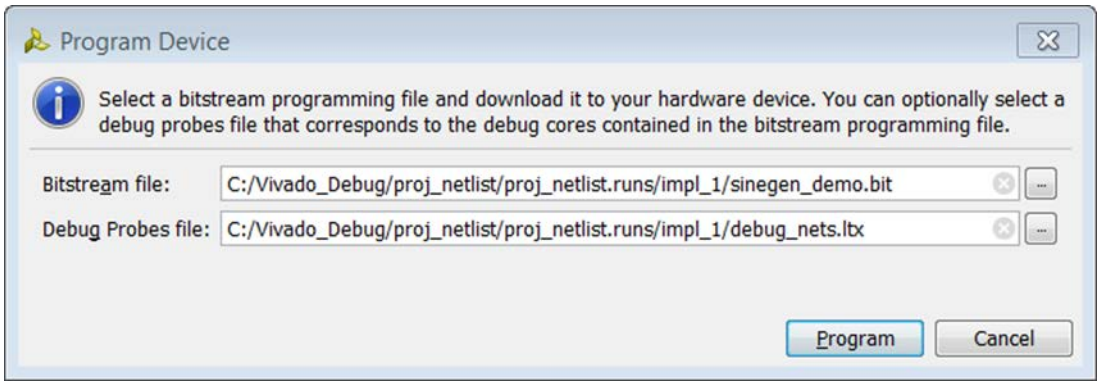

Program Device
Select a bitstream programming file and download it to your hardware device. You can optionally select a debug probes file that corresponds to the debug cores contained in the bitstream programming file.
Bitstream file:
C:/Vivado_Debug/proj_netlist/proj_netlist.runs/impl_1/sinegen_demo.bit
Debug Probes file:
C:/Vivado_Debug/proj_netlist/proj_netlist.runs/impl_1/debug_nets.ltx
Program
Cancel

Figure 65: Select Bitstream file to download for Lab 1

CAUTION! *The file paths of the bitstream to be programmed will be different for different labs. Ensure that the relative paths are correct.*

***Note**: Wait for the program device operation to complete. This may take few minutes.*

11. Ensure that an ILA core was detected in the **Hardware** panel of the **Debug** view.

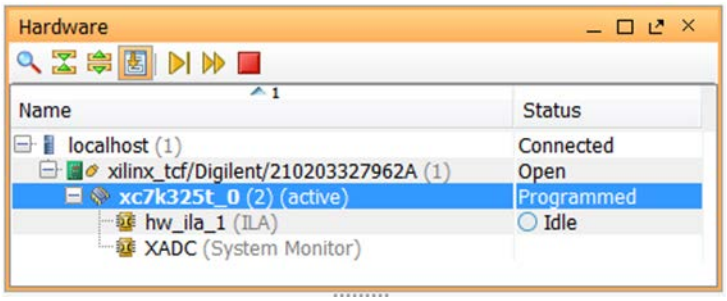


Figure 66: ILA Core Detection

12. The Integrated Logic Analyzer window opens.

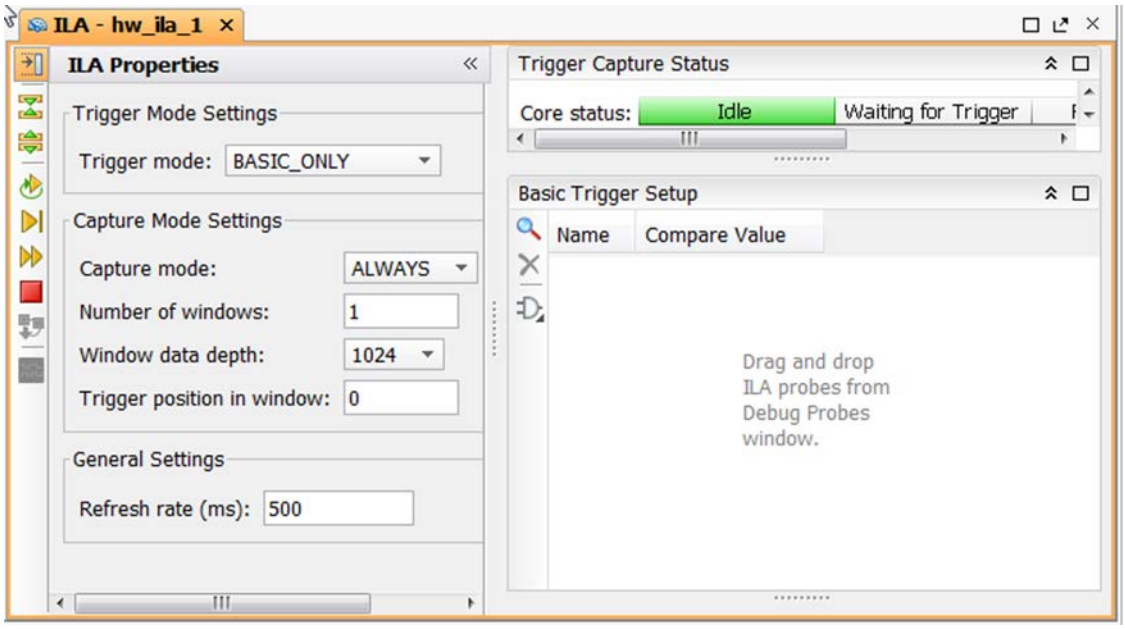


Figure 67: The Vivado Integrated Logic Analyzer window

Verifying Sine Wave Activity

13. Click **Run Trigger Immediate** to trigger and capture data immediately as shown in shown in the following figure.

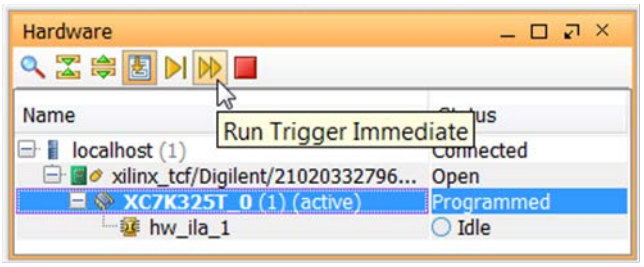


Figure 68: Run Trigger Immediate Button

14. In the **Waveform** window, verify that there is activity on the 20-bit sine signal as shown in the following figure.

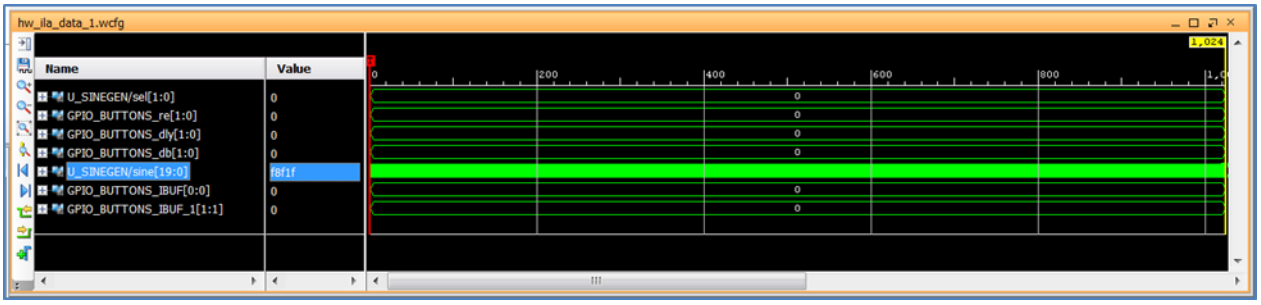


Figure 69: Output Sine Wave Displayed in Digital Format

Displaying the Sine Wave

15. Right-click **U_SINEGEN/sine[19:0]** signals, and select **Waveform Style >Analog** as shown in the following figure.

TIP: *The waveform does not look like a sine wave. This is because you must change the radix setting from Hex to Signed Decimal, as described in the following subsection.*

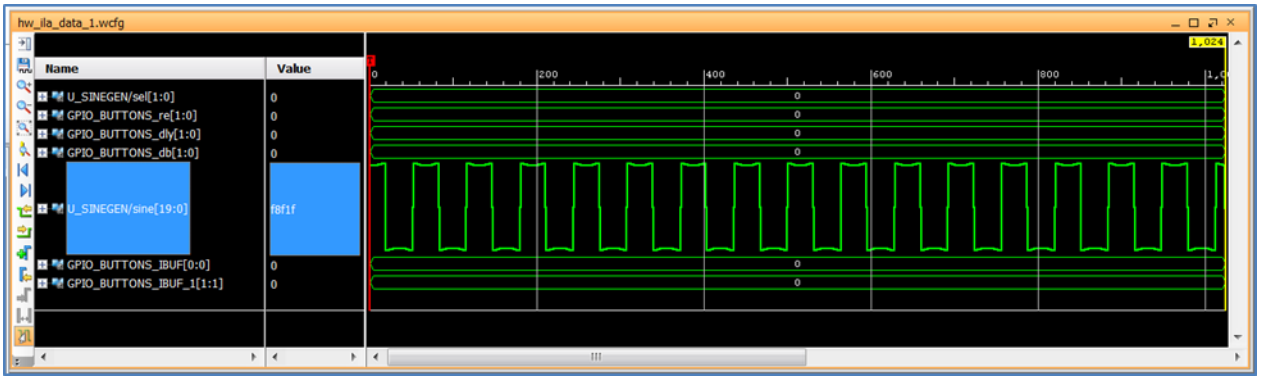


Figure 70: Output Sine Wave Displayed in Analog Format, High Frequency - 1

16. Right-click **U_SINEGEN/sine[19:0]** signals, and **select Radix > Signed Decimal**. You should now be able to see the high frequency sine wave as shown in the following figure instead of the square wave.

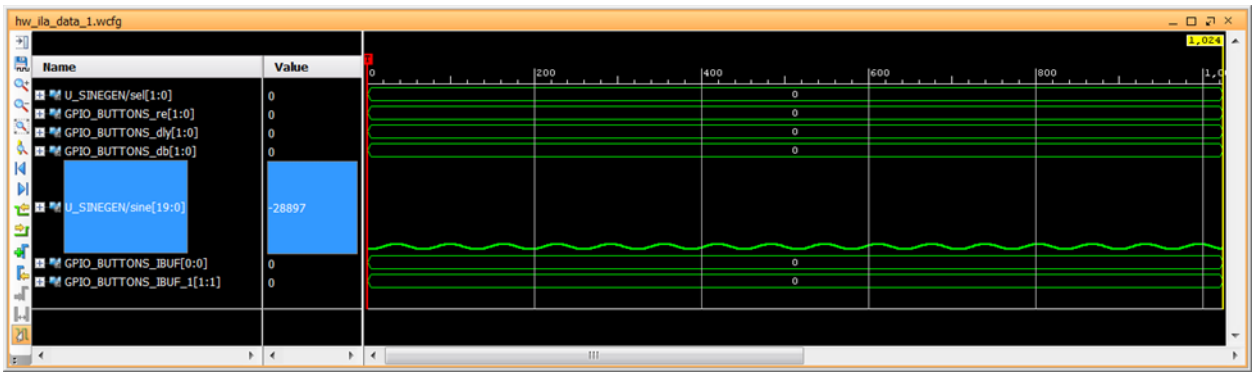


Figure 71: Output Sine Wave Displayed in Analog Format - High Frequency -2

Correcting Display of the Sine Wave

To view the mid, and low frequency output sine waves, perform the following steps:

17. Cycle the sine wave sequential circuit by pressing the GPIO_SW_E push button as shown in the following figure.

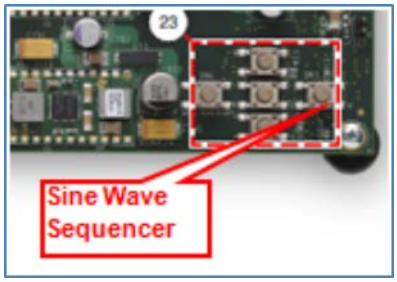


Figure 72: Sine Wave Sequencer Push Button

18. Click **Run Trigger Immediately** again to see the new sine selected sine wave. You should see the mid frequency as shown in the following figure. Notice that the `sel` signal also changed from 0 to 1 as expected.

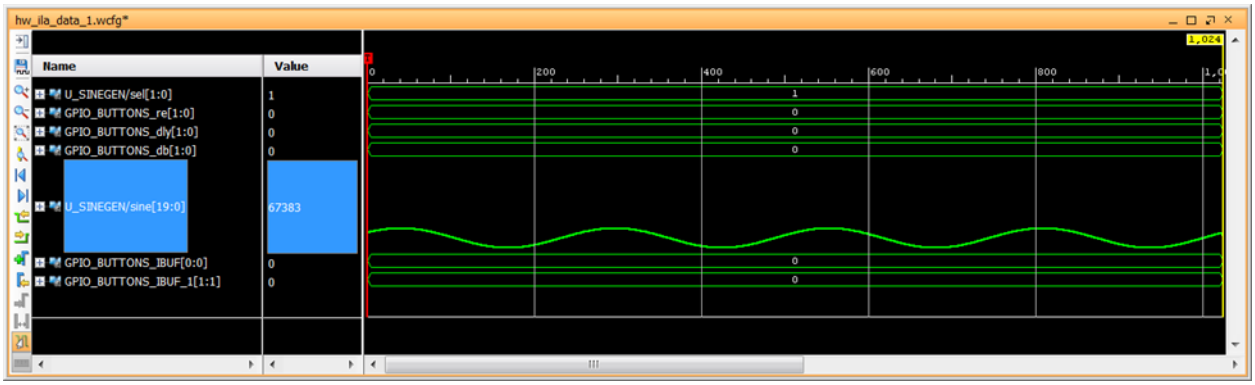


Figure 73: Output Sine Wave Displayed in Analog Format – Mid Frequency

19. Repeat step 17 and 18 to view other sine wave outputs.

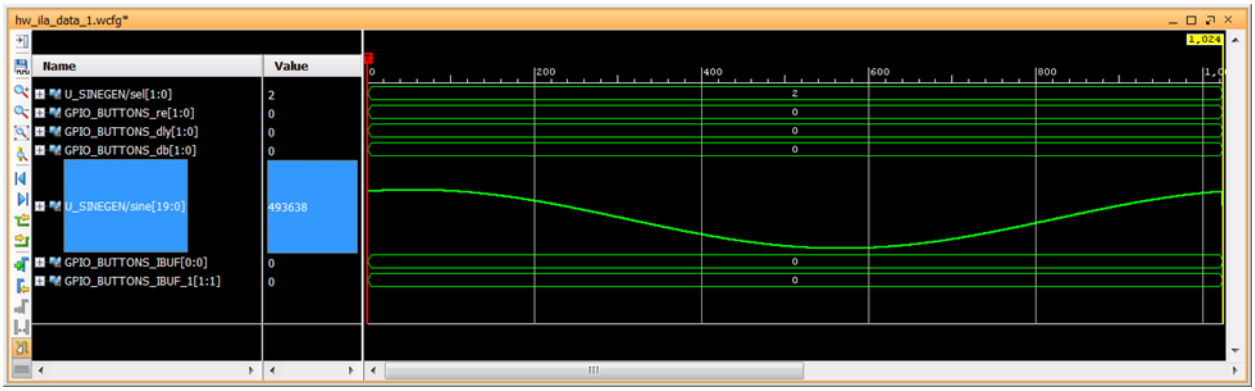


Figure 74: Output Sine Wave Displayed in Analog Format – Low Frequency

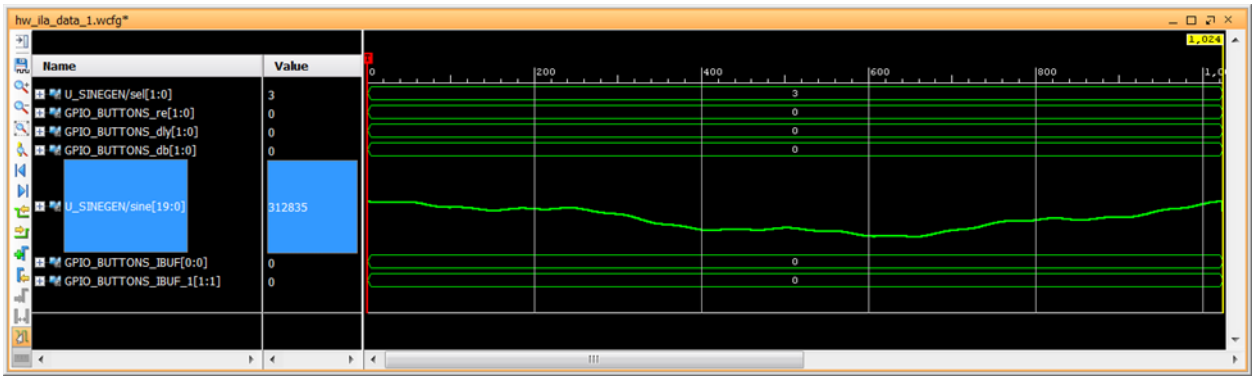


Figure 75: Output Sine Wave Displayed in Analog Format – Mixed Frequency

Note: As you sequence through the sine wave selections, you may notice that the LEDs do not light up in the expected order. You will debug this in the next section of this tutorial. For now, verify for each LED selection, that the correct sine wave displays. Also, note that the signals in the **Waveform** window have been re-arranged in the previous three figures.

Step 2: Debugging the Sine Wave Sequencer State Machine (Optional)

As you were correcting the sine wave display, the LEDs might not have lit up in sequence as you pressed the Sine Wave Sequencer button. With each push of the button, there should be a single, cycle-wide pulse on the GPIO_BUTTONS_re[1] signal. If there is more than one, the behavior of the LEDs becomes irregular. In this section of the tutorial, use Vivado logic analyzer to probe the sine wave sequencer state machine, and to view and repair the root cause of the problem.

Before starting the actual debug process, it is important to understand more about the sine wave sequencer state machine.

Sine Wave Sequencer State Machine Overview

The sine wave sequencer state machine selects one of the four sine waves to be driven onto the sine signal at the top-level of the design. The state machine has one input and one output. The following figure shows the schematic elements of the state machine. Refer to this diagram as you read the following description and as you perform the steps to view and repair the state machine glitch.

- The input is a scalar signal called “button”. When the button input equals “1”, the state machine advances from one state to the next.
- The output is a 2-bit signal vector called “Y”, and it indicates which of the four sine wave generators is selected.

The input signal button connects to the top-level signal GPIO_BUTTONS_re[1], which is a low-to-high transition indicator on the Sine Wave Sequencer button. The output signal Y connects to the top-level signal, sineSel, which selects the sine wave.

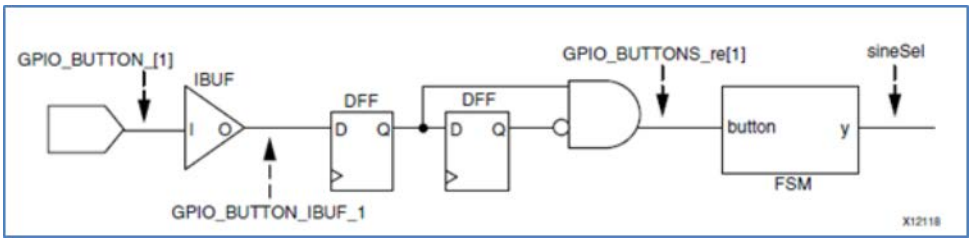


Figure 76: Sine Wave Sequencer Button Schematic

Viewing the State Machine Glitch

You cannot troubleshoot the issue you identified above by connecting a debug probe to the GPIO_BUTTON [1] input signal itself. The GPIO_BUTTON [1] input signal is a PAD signal that is not directly accessible from the FPGA fabric. Instead, you must trigger on low-to-high transitions (rising edges) on the GPIO_BUTTON_IBUF signal, which is connected to the output of the input buffer of the GPIO_BUTTON [1] input signal.

As described earlier, the glitch reveals itself as multiple low-to-high transitions on the GPIO_BUTTONS_1_IBUF signal, but it occurs intermittently. Because it could take several button presses to detect it, you will now set up the Vivado logic analyzer tool to Repetitive Trigger Run Mode. This setting makes it easier to repeat the button presses and look for the event in the Waveform viewer.

1. Open the **Debug Probes** window if not already open by selecting **Window > Debug Probes** from the menu.
2. In the **ILA Properties** window scroll down to the link marked **To view editable ILA Properties: Open ILA Dashboard** and set the following:
   a. **Trigger Mode** to **BASIC_ONLY**
   b. **Capture Mode** to **BASIC**
   c. **Window Data Depth** to **1024**
   d. **Trigger position** to **512**
   e. Drag and drop the **GPIO_BUTTONS_IBUF_1 [1:1]** bit from the **Debug Probes** window, in the **Basic Trigger Setup** dialog box to the **Name** field. Change the **Compare Value** field to **R** by clicking in the **Compare Value** column and typing the value **R** in the **Value** field, as shown in the following figure.

Figure 77: Setting Trigger Conditions

CAUTION! *For different labs the GPIO_BUTTONS_IBUF may show up differently. This may show up as two individual bits or two bits lumped together in a bus. Ensure that you are using bit 1 of this bus to set up your trigger condition. For example in case of a two-bit bus, you will set the* ***Value*** *field in the* ***Compare Value*** *dialog box to* ***RX****.*

CAUTION! *The* ***ILA properties*** *window may look slightly different for different labs.*

3. Select the **hw_ila_data_1.wcfg** tab or the **waveform** tab. Source the `rt.tcl` file in the Tcl Console by typing `source rt.tcl`. This Tcl command performs the following tasks:

- Arms the trigger.
- Waits for the trigger.
- Uploads and displays waveforms.

4. On the KC705 board, press the Sine Wave Sequencer button until you see multiple transitions on the GPIO_BUTTONS_1_IBUF signal (this could take 10 or more tries). This is a visualization of the glitch that occurs on the input. An example of the glitch is shown in the following two figures.

CAUTION! *You may have to repeat steps 3 and 4 repeatedly to see the glitch. Once you can see the glitch, you may observe signal glitches are not at exactly the same location as shown in the figure below.*

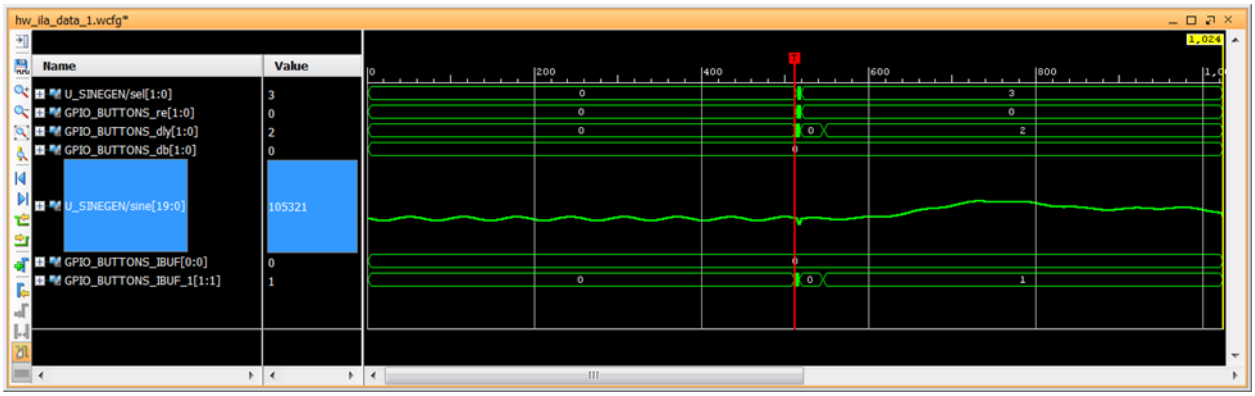


Figure 78: GPIO_BUTTONS_BUF1 Signal Glitch

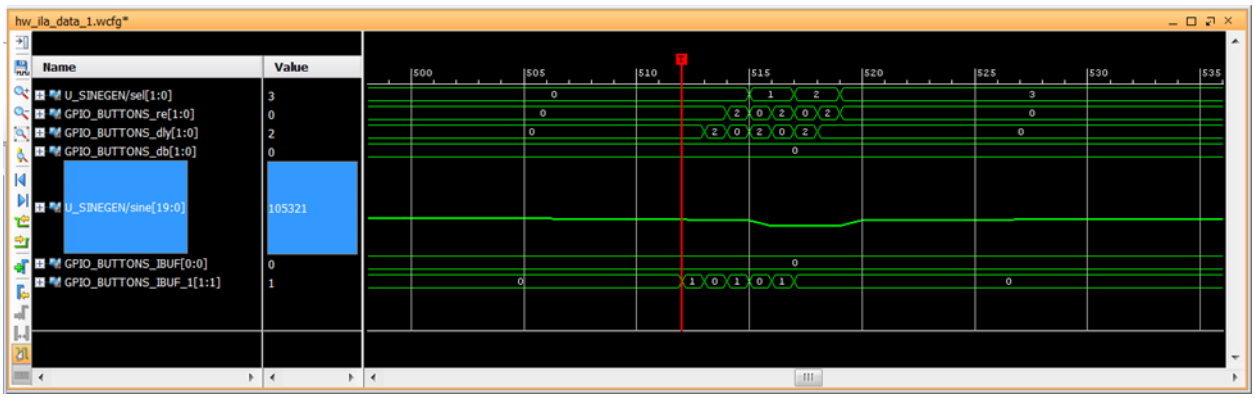


Figure 79: GPIO Buttons_1_re Signal Glitch magnified

Fixing the Signal Glitch and Verifying the Correct State Machine Behavior

The multiple transition glitch or "bounce" occurs because the mechanical button is making and breaking electrical contact just as you press it. To eliminate this signal bounce, a "de-bouncer" circuit is required.

5. Enable the de-bouncer circuit by setting DIP switch position on the KC705 board (labeled De-bounce Enable in Figure 1: KC705 Board Showing Key Components) to the **ON** or **UP** position.
6. Source the `rt.tcl` file again by typing `source rt.tcl` in the Tcl Console, and:
    - Ensure that you no longer see multiple transitions on the GPIO_BUTTON_re[1] signal on a single press of the Sine Wave Sequencer button.
    - Verify that the state machine is working correctly by ensuring that the sineSel signal transitions from 00 to 01 to 10 to 11 and back to 00 with each successive button press.

Verifying the VIO Core Activity (Only applicable to Lab 3)

1. From the **Program and Debug** section in Flow Navigator, click **Open Hardware Manager**.

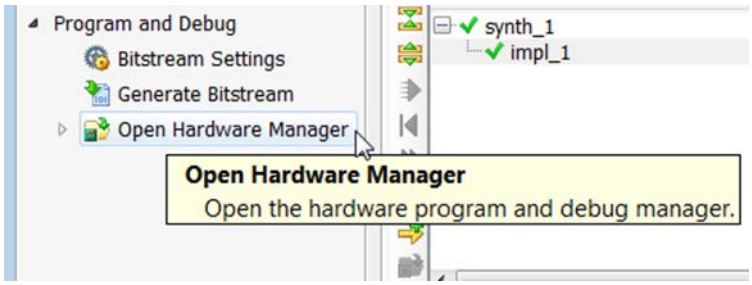


Figure 80: Open Hardware Manager

2. The **Hardware Manager** window opens. Click **Open a new hardware target**.

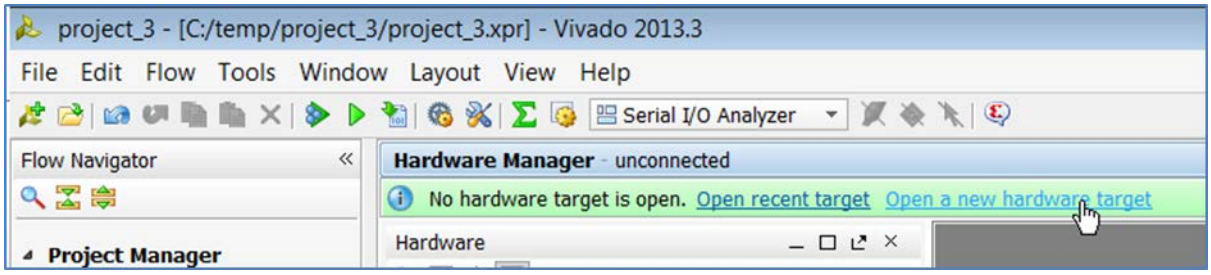

project_3 - [C:/temp/project_3/project_3.xpr] - Vivado 2013.3
File Edit Flow Tools Window Layout View Help
Serial I/O Analyzer
Flow Navigator
Project Manager
Hardware Manager - unconnected
No hardware target is open. Open recent target Open a new hardware target
Hardware

Figure 81: Connect to a new hardware target

3. The **Open New Hardware Target** wizard opens. Click **Next**.
4. In the **Hardware Server Settings** page, type the name of the server (or select **Local server** if the target is on the local machine) in the **Connect to** field.
5. Ensure that you are connected to the right target by selecting the target from the **Hardware Targets** page. If there is only one target, that target is selected by default. Click **Next**.
6. In the **Set Hardware Target Properties** page, click **Next**.
7. In the **Open Hardware Target Summary** page, verify that all the information is correct, and click **Finish**.
8. Program the device by selecting and right-clicking the device in the **Sources** window and then selecting **Program Device**.

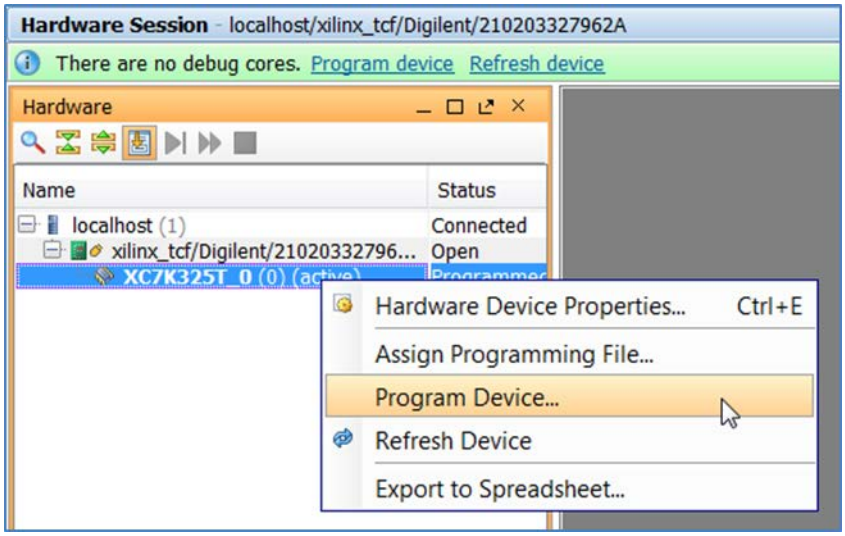


Figure 82: Program FPGA

9. In the **Program Device** dialog box, ensure that the bit file to be programmed is correct. Click **OK**.

Figure 83: Program Device with the sinegen_demo_inst_vio.bit file

10. After the FPGA device is programmed, you see the VIO and the ILA core in the **Hardware** window.

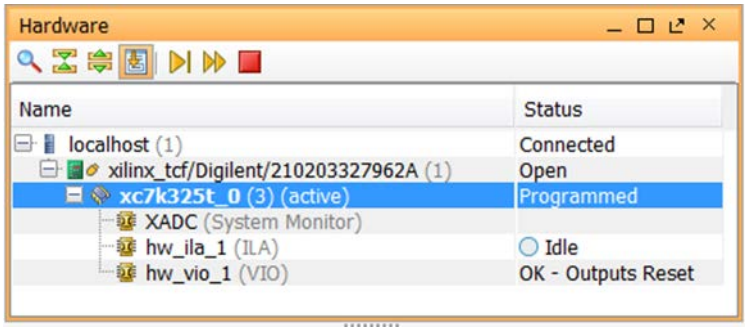


Figure 84: The ILA and VIO cores in the Hardware window

11. Click **Run Trigger Immediate** to capture the data immediately.

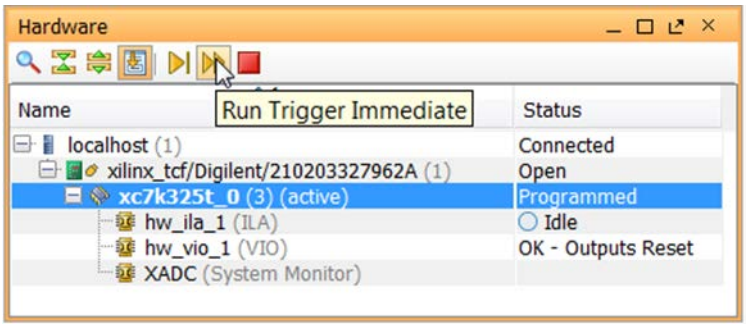


Figure 85: Run Trigger Immediate

12. Make sure that there is activity on the sine [19:0] signal.
13. Select the sine signal in the **Waveform** window, right-click and select **Waveform Style > Analog**.
14. Select the sine signal in the **Waveform** window again, right-click and select **Radix > Signed Decimal**. You should be able to see the sine wave in the **Waveform** window.

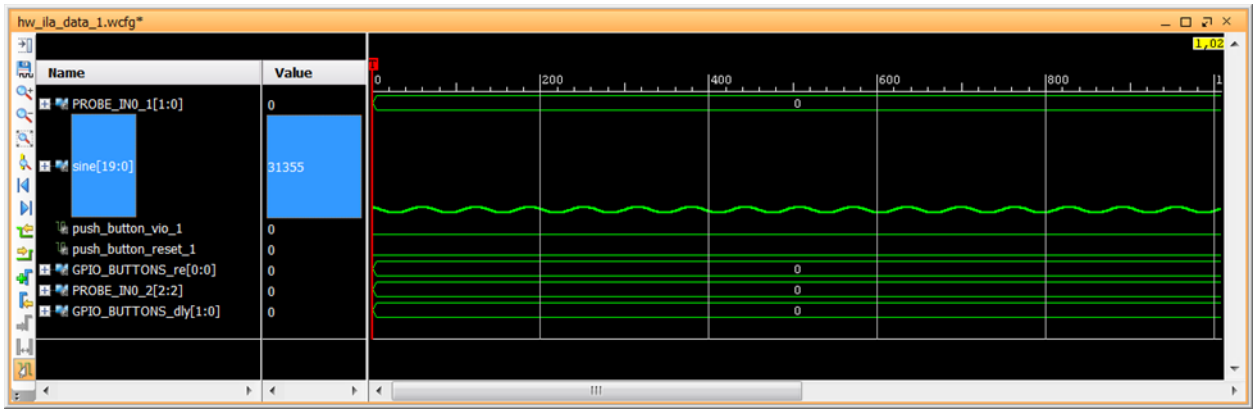


Figure 86: Sine wave after modifying the properties of the sine [19:0] signal

15. Instead of using the GPIO_SW push button to cycle through each different sine wave output frequency, you are going to use the virtual "push_button_vio" toggle switch from the VIO core.

16. If the **Debug Probes** window is not open already, open it by selecting **Window > Debug Probes** from the menu.

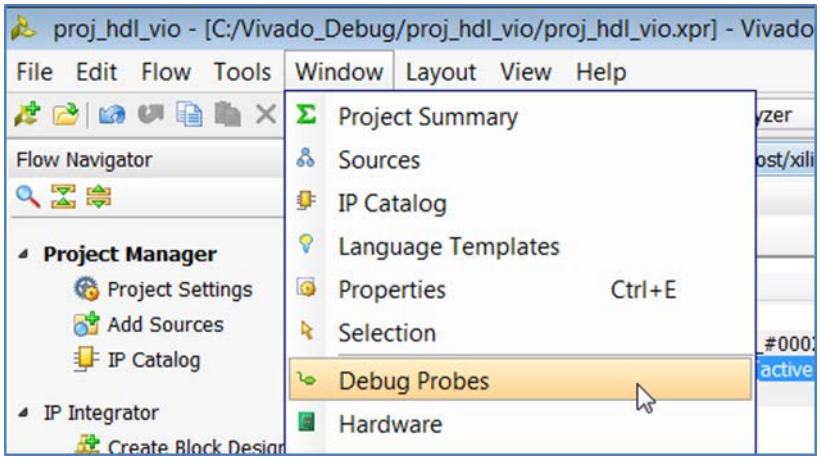


Figure 87: Open Debug Probes window

The **Debug Probes** window shows the probes available in the ILA as well as the VIO cores.

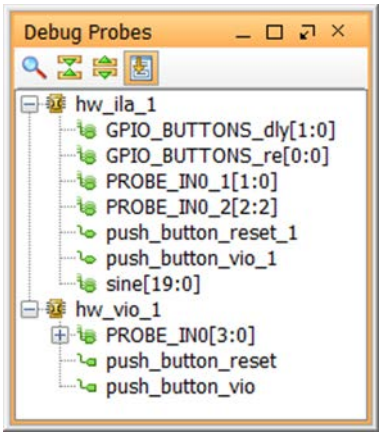


Figure 88: Debug probes window showing probes for ILA as well as VIO

Also, note that separate tabs are available for the ILA as the VIO cores as shown below.

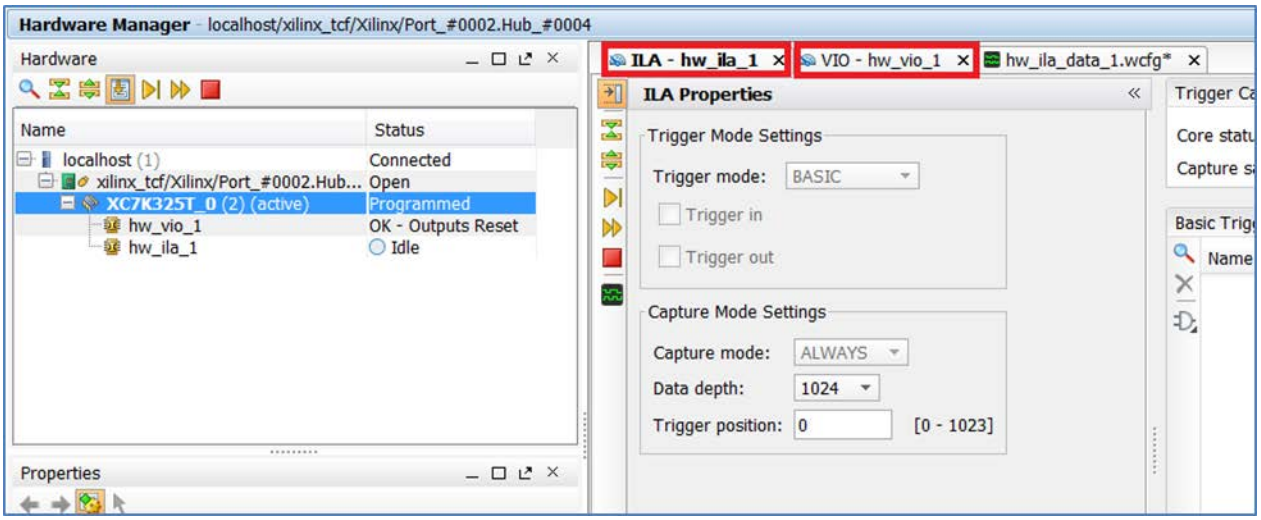


Figure 89: Tabs for ILA and VIO

17. In the **VIO Probes** window, from the hw_vio_1, select all the probes, then drag-and-drop these probes in the VIO core tab. Note the initial values of all the probes.

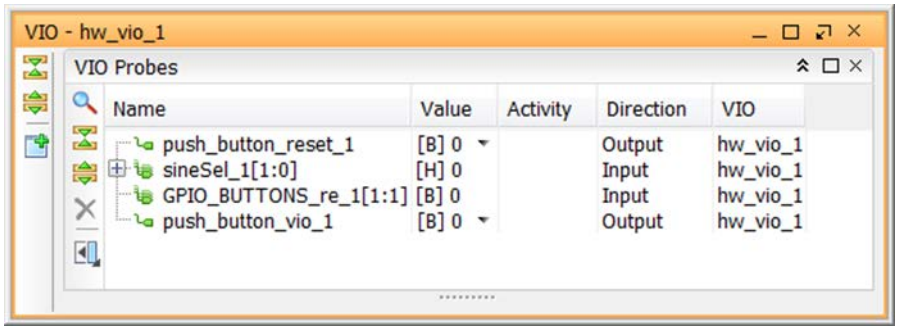


Figure 90: Drag-and-drop VIO probes from the Debug Probes window into VIO core tab

18. Set the "push_button_reset" output probe by right-clicking **push_botton_reset** and select **Toggle Button**. This will toggle the output driver from logic from '0' to '1' to '0' as you click. It is similar to the actual push button behavior, though there is no bouncing mechanical effect as with a real push button switch.

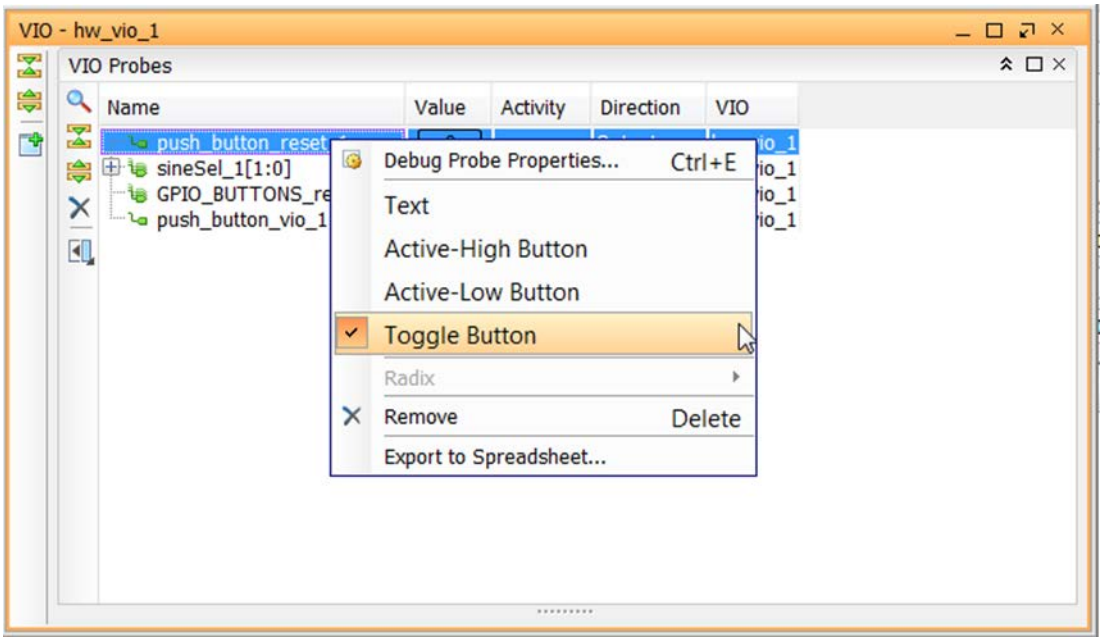


Figure 91: Toggle the push_button_reset signal

The **Value** field for push_button_reset is highlighted. Click in the **Value** field to change its value to **1**.

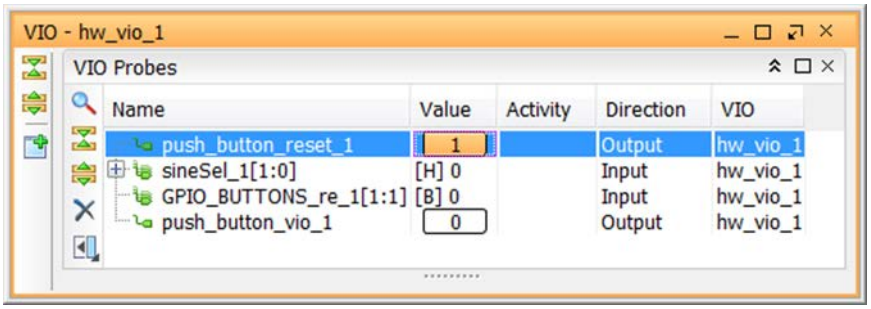


Figure 92: Toggle the value of push_button_reset

19. Follow the step above to change the push_button_vio to Toggle button as well.

20. Set these two bits of the "sineSel" input probe by right-clicking **PROBE_IN0[0] and PROBE_IN0[1]** and selecting **LED**.

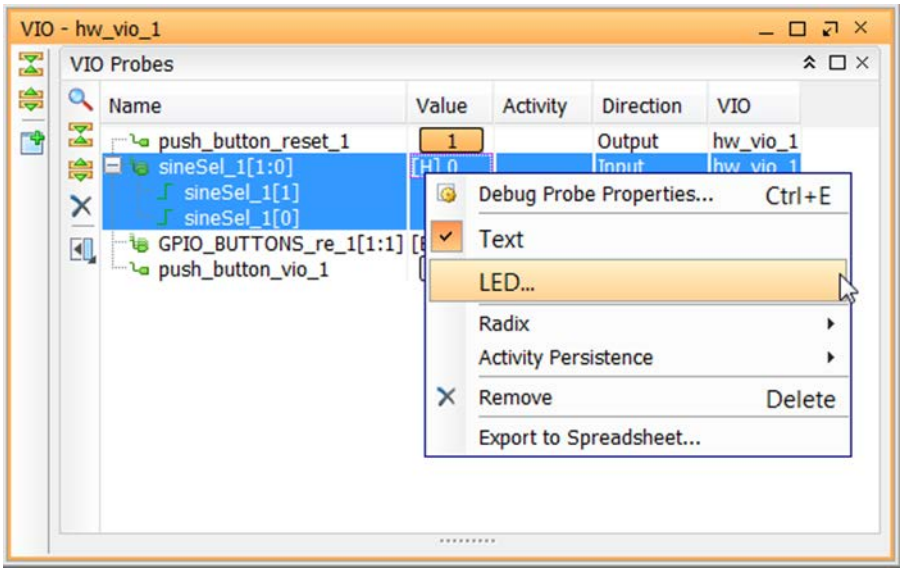


Figure 93: Change sineSel to LED

21. In the **Select LED Colors** dialog box, pick the **Low Value Color** and the **High Value Color** of the LEDs as you desire and click **OK**.

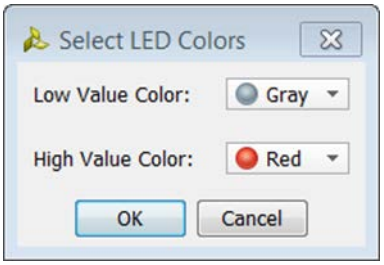


Figure 94: Pick the Low Value and High Value color of the LEDs

22. When finished, your **VIO Probes** window in the Hardware Manager should look similar to the following figure.

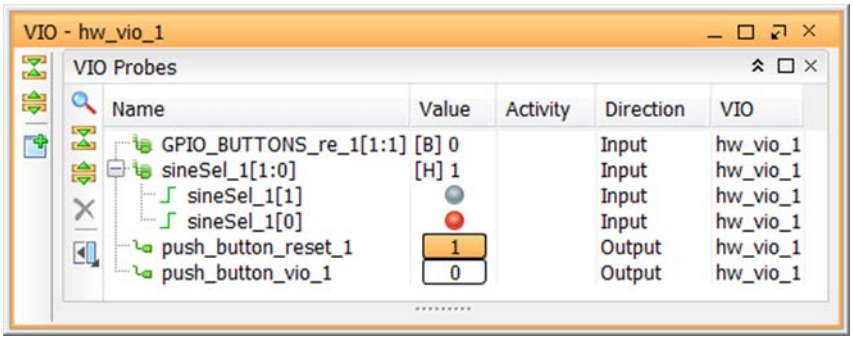


Figure 95: Input and Output VIO Signals Displayed

23. To cycle through each different sine wave output frequency using the virtual "push_button_vio" from the VIO core, follow the following simple steps:
    a. Toggle the value of the "push_button_vio" output driver from 0 to 1 to 0 by clicking on the logic displayed under the **Value** column. You will notice the sineSel LEDs changed accordingly – 0, 1, 2, 3, 0, etc…
    b. Select **hw_ila_1** in the **Hardware** window and click **Run Trigger** to capture and display the selected sine wave signal from the previous step.

Lab 5: Using Vivado Serial Analyzer to Debug Serial Links

Introduction

The Serial I/O analyzer is used to interact with IBERT Version 3.0 or later debug IP cores contained in a design. It is used to debug and verify issues in high speed serial I/O links.

The Serial I/O Analyzer has several benefits as listed below:

- Tight integration with Vivado® IDE.
- Ability to script during netlist customization/generation and serial hardware debug.
- Common interface with the Vivado Integrated Logic Analyzer.

The customizable LogiCORE™ IP Integrated Bit Error Ratio Tester (IBERT) core for 7 series FPGA GTX transceivers is designed for evaluating and monitoring the GTX transceivers. This core includes pattern generators and checkers that are implemented in FPGA logic, and provides access to ports and the dynamic reconfiguration port attributes of the GTX transceivers. Communication logic is also included to allow the design to be run time accessible through JTAG.

In the course of this tutorial, you:

- Create, customize, and generate an Integrated Bit Error Ratio Tester (IBERT) core design in the Vivado Integrated Design Suite.
- Interact with the design using Serial I/O Analyzer. This includes connecting to the target KC705 board, configuring the device, and interacting with the IBERT/Transceiver IP cores.
- Perform a sweep test to optimize your transceiver channel and to plot data using the IBERT sweep plot GUI feature.

Design Description

You can customize the IBERT core and use it to evaluate and monitor the functionality of transceivers for a variety of Xilinx devices. The focus for this tutorial is on Kintex®-7 GTX transceivers. Accordingly, the KC705 target board is used for this tutorial.

The following figure shows a block diagram of the interface between the IBERT Kintex-7 GTX core interfaces with Kintex-7 transceivers.

- **DRP Interface and GTX Port Registers**: IBERT provides you with the flexibility to change GTX transceiver ports and attributes. Dynamic reconfiguration port (DRP) logic is included, which allows the runtime software to monitor and change any attribute in any of the GTX transceivers included in the IBERT core. When applicable, readable and writable registers are also included. These are connected to the ports of the GTX transceiver. All are accessible at run time using the Vivado Logic Analyzer tool.
- **Pattern Generator**: Each GTX transceiver enabled in the IBERT design has both a pattern generator and a pattern checker. The pattern generator sends data out through the transmitter.
- **Error Detector**: Each GTX transceiver enabled in the IBERT design has both a pattern generator and a pattern checker. The pattern checker takes the data coming in through the receiver and checks it against an internally generated pattern.

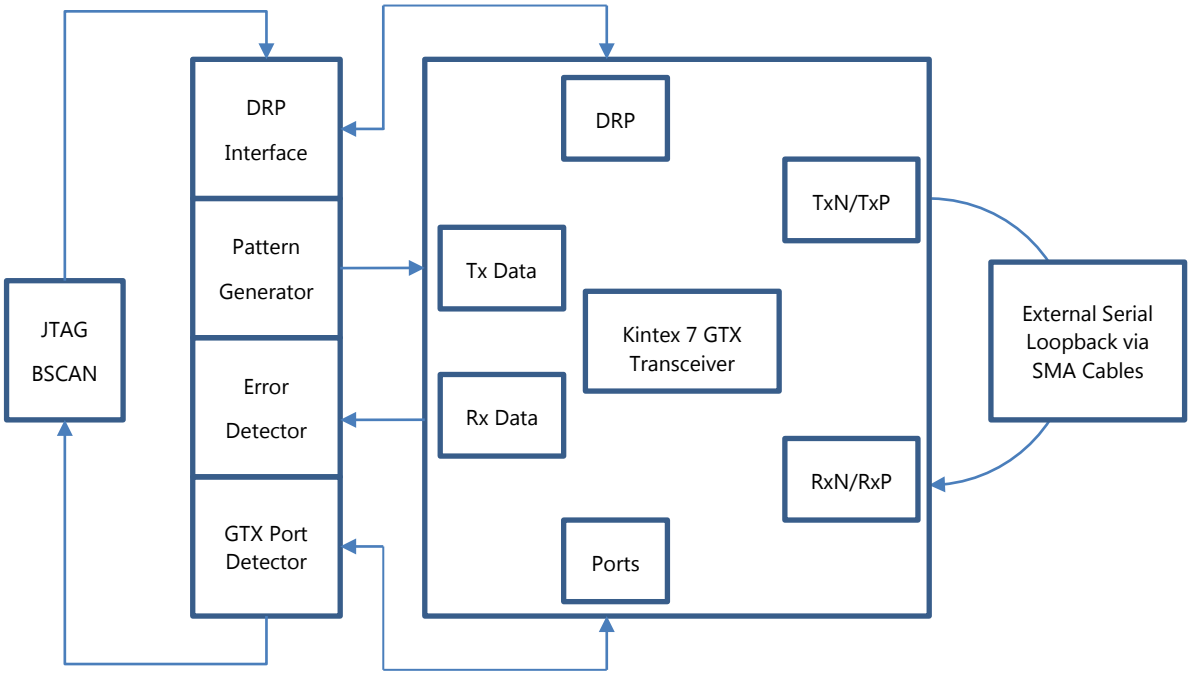


Figure 96: IBERT Design Flow

Step 1: Creating, Customizing, and Generating an IBERT Design

To create a project, use the New Project wizard to name the project, to add RTL source files and constraints, and to specify the target device.

1. Invoke the Vivado IDE.
2. In the **Getting Started** screen, click **Create New Project** to start the **New Project** wizard. Click **Next**.
3. In the **Project Name** page, name the new project **ibert_tutorial** and provide the project location (`C:/ibert_tutorial`). Ensure that **Create Project Subdirectory** is selected. Click **Next**.
4. In the **Project Type** page, specify the **Type of Project** to create as **RTL Project.** Click **Next**.
5. In the **Add Sources** page click **Next**.
6. In the **Add Existing IP** page, click **Next**.
7. In the **Add Constraints** page, click **Next**.
8. In the **Default Part** page, select **Boards** and then select **Kintex-7 KC705 Evaluation Platform**. Click **Next**.
9. Review the **New Project Summary** page. Verify that the data appears as expected, per the steps above. Click **Finish**.

   ***Note**: It might take a moment for the project to initialize.*

Step 2: Adding an IBERT core to the Vivado Project

1. In the Flow Navigator click **IP Catalog**. The IP Catalog opens.

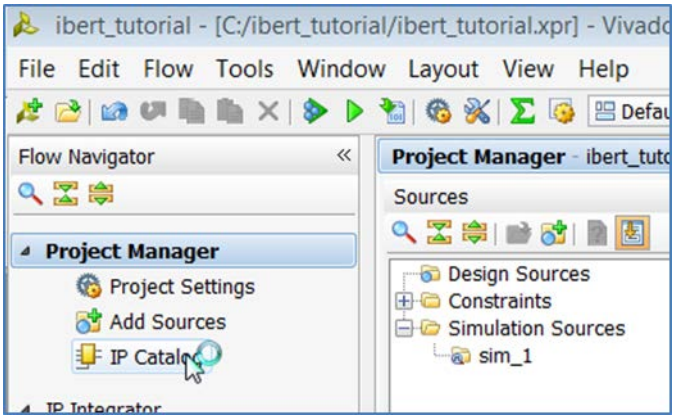


Figure 97: Opening the Vivado IP Catalog

2. In the search field of the IP Catalog type **IBERT**, to display the IBERT 7 Series GTX IP.

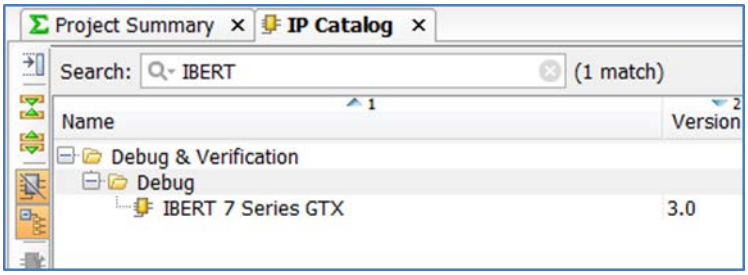


Figure 98: Instantiating the IBERT IP from the Vivado IP Catalog

3. Double-click **IBERT 7 Series GTX IP**. This brings up the customization GUI for the IBERT.
4. In the **Customize IP** dialog box, choose the following options in the **Protocol Definition** tab:
   a. Type the name of the component in the **Component Name** field. In this case, leave the name as the default name, **ibert_7series_gtx_0**.
   b. Ensure that the **Silicon Version** is selected as **General ES/Production**.
   c. Ensure that the **Number of Protocols** option is set to **1**.
   d. Change the **LineRate (Gbps)** to **8**.
   e. Change **DataWidth** to **40**.
   f. Change **Refclk (MHz)** to **125**.
   g. Ensure that the **Quad Count** is set to **2**.
   h. Ensure **Quad PLL** box is selected.

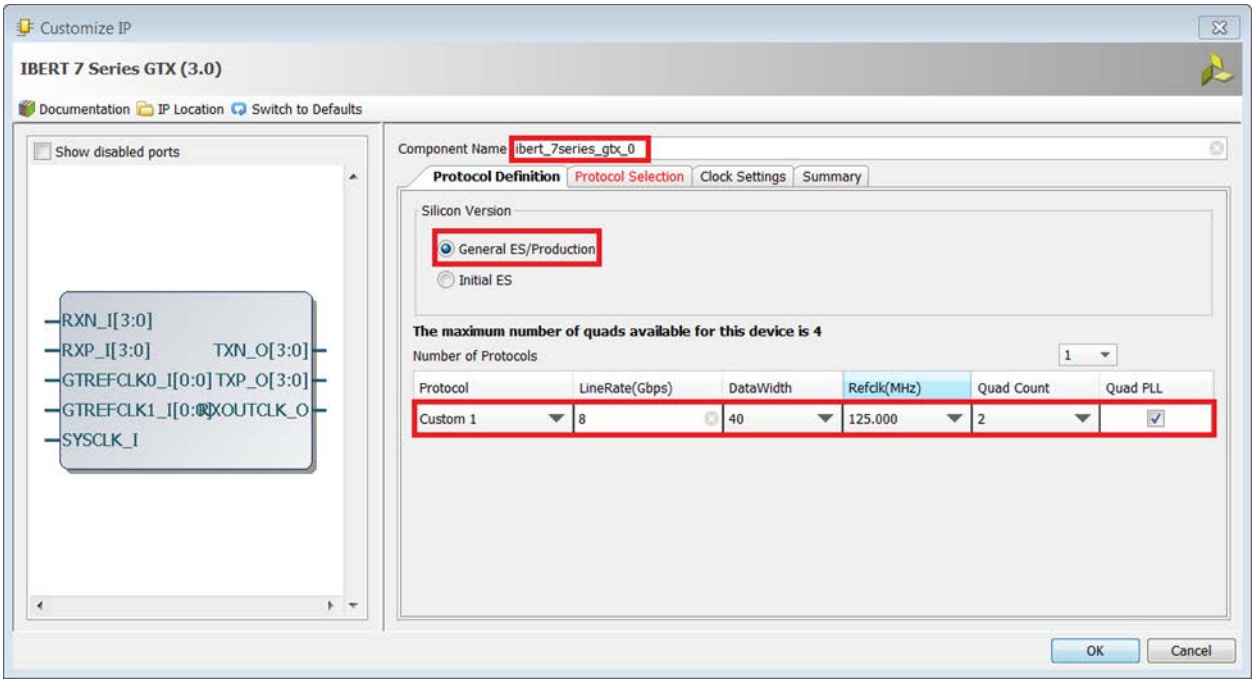


Figure 99: Setting the Protocol Definition on the IBERT Core

5. Under the **Protocol Selection** tab, update the following selections:
   a. For GTX Location QUAD_117, in the **Protocol Selected** column, click the pull-down menu and select **Custom 1 / 8 Gbps**. This should automatically populate **Refclk Selection** to **MGTREFCLK0 117** and **TXUSRCLK Source** to **Channel 0**.
   b. For GTX Location QUAD_118, do the following:
      i. In the **Protocol Selected** column, click the pull-down menu and select **Custom 1 / 8 Gbps.**
      ii. In the **Refclk Selection** column, change the value to **MGTREFCLK0 117**.
      iii. In the **TXUSRCLK Source** column, change the value to **Channel 0**.

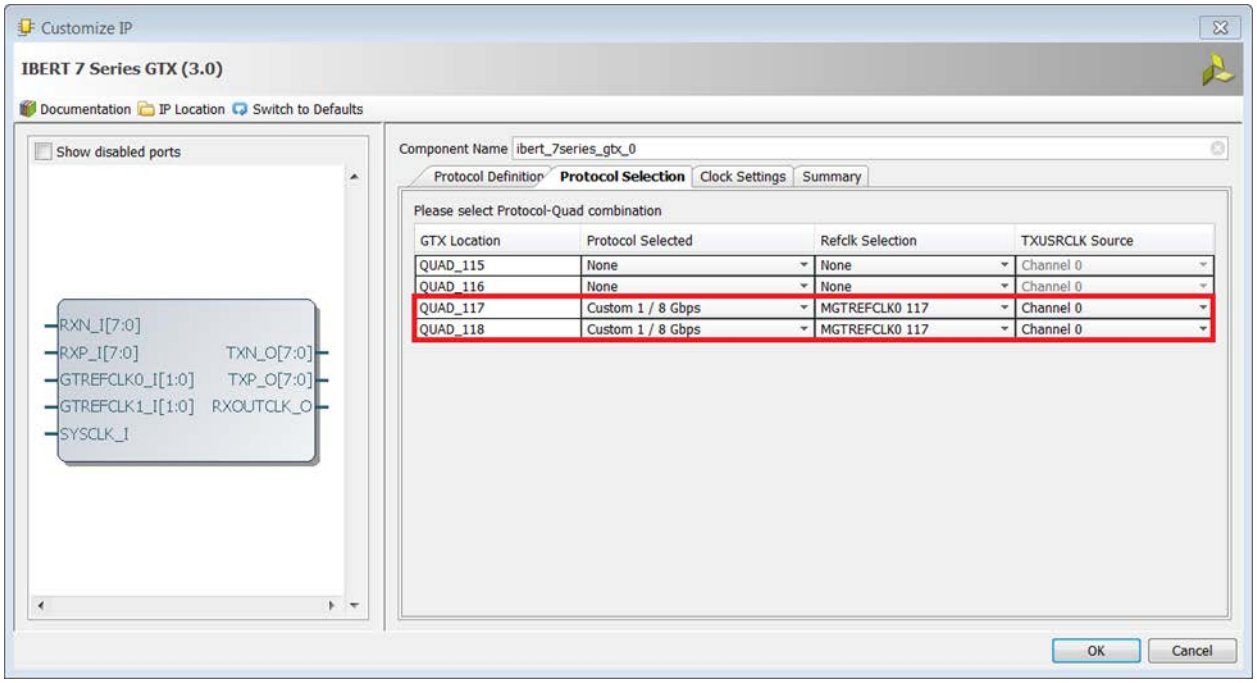


Figure 100: Setting the Protocol Selection on the IBERT Core

6. Click the **Clock Settings** tab and make the following changes for both QUAD_117 and QUAD_118:
   a. Leave the **Source** column at its default value of **External**.
   b. Change the **I/O Standard** column to **DIFF SSTL15**.
   c. Change the **P Package Pin** to **AD12**.
   d. Change the **N Package Pin** to **AD11**.
   e. Leave the **Frequency(MHz)** at its default value of **200.00**.

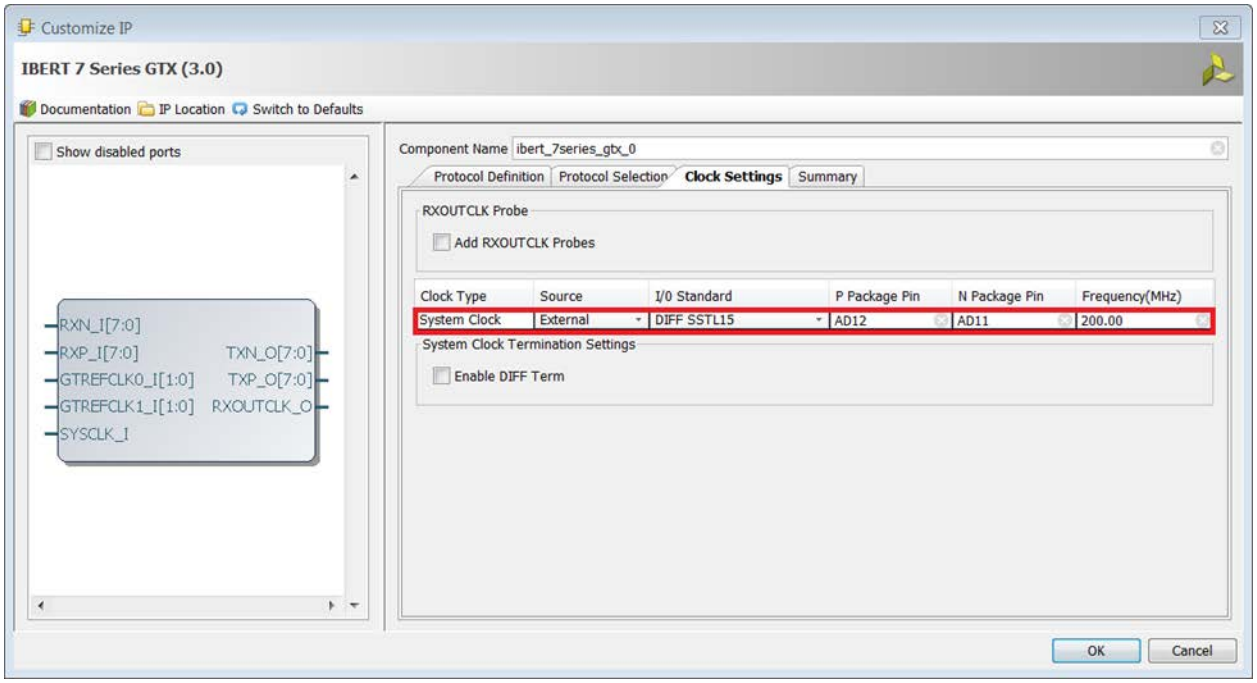


Figure 101: Specifying clock settings for the IBERT Core

7. Click the **Summary** tab and ensure that the content matches the following figure. Click **OK**.

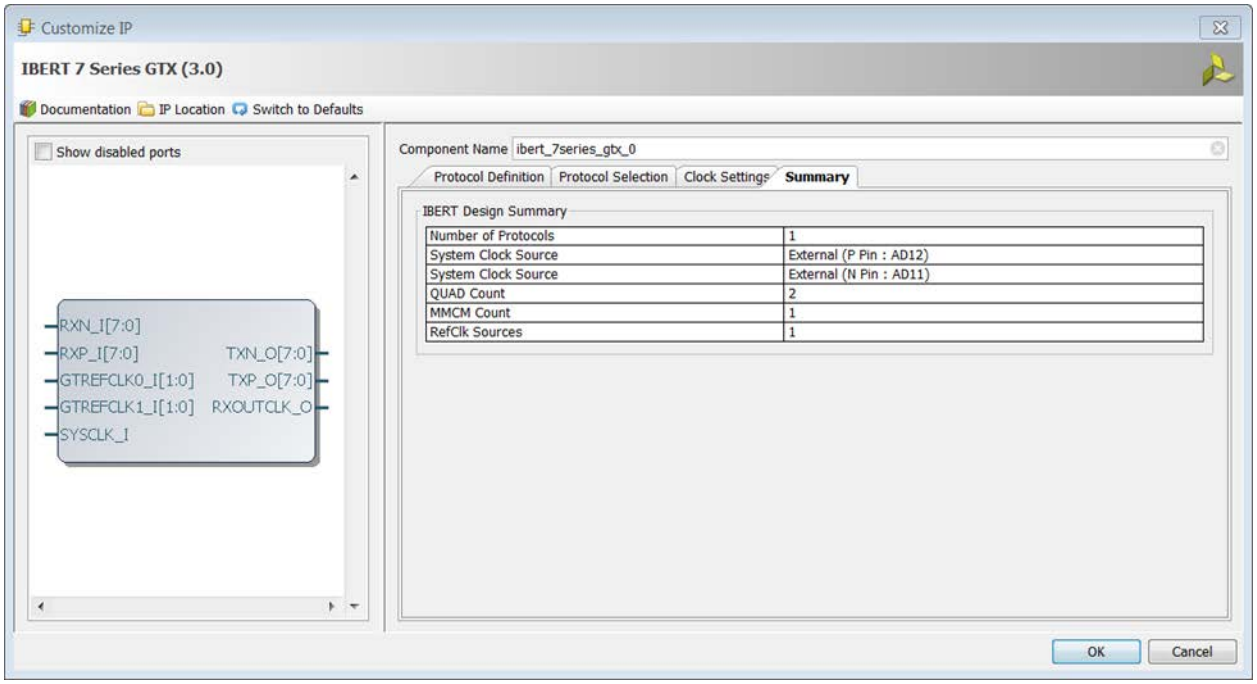


Figure 102: IBERT Core Summary Page

8. When the **Generate Output Products** dialog box opens, click **Generate**.

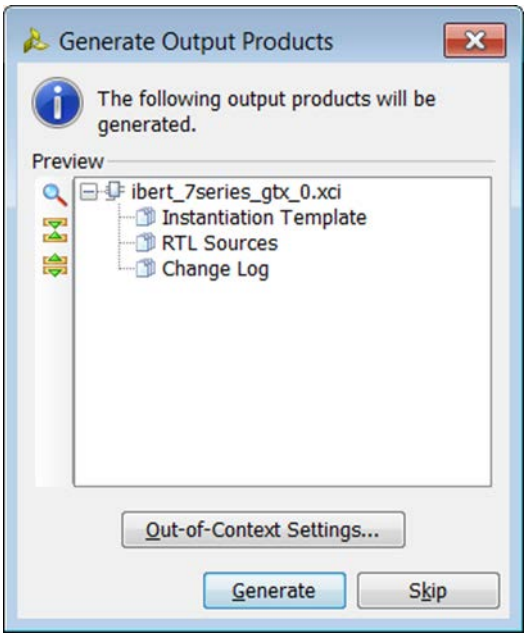


Figure 103: Generate Output Products

9. In the **Project Manager** window, right-click the IP, and select **Open IP Example Design**.

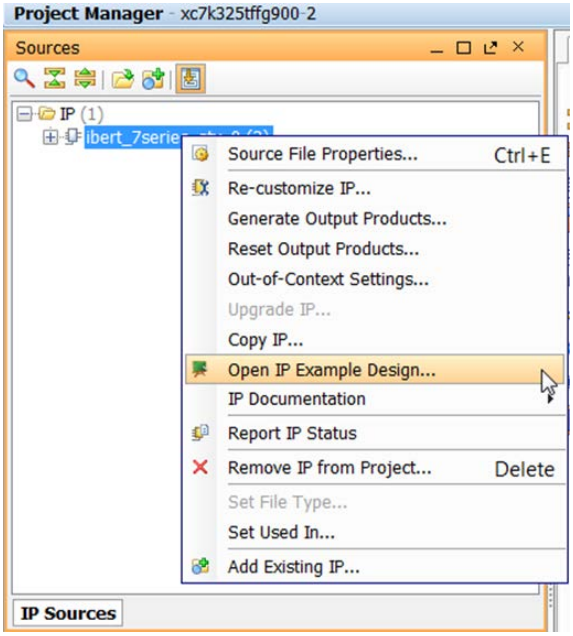


Figure 104: Open Example IP Design Menu Item

10. In the **Open IP Example Design** dialog box, ensure that the **Overwrite existing example project** is selected and click **OK**.

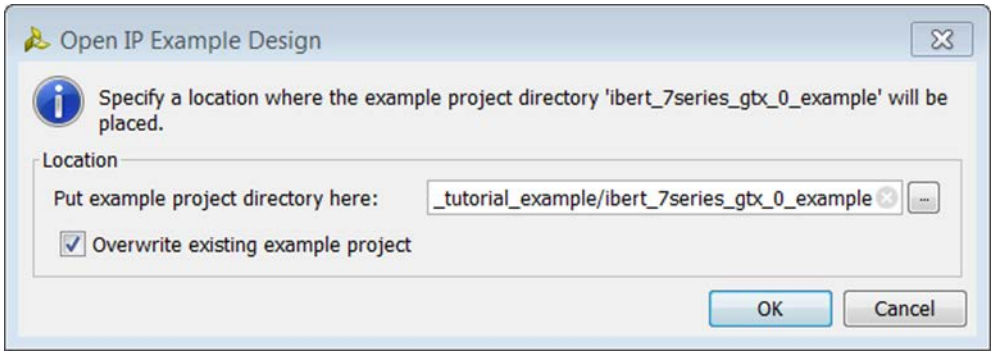


Figure 105: Open IP Example Design dialog

Step 3: Synthesize, Implement and Generate Bitstream for the IBERT design

1. Click **Generate Bitstream** in the Flow Navigator. When the **No Implementation Results Available** dialog box appears. Click **Yes**.

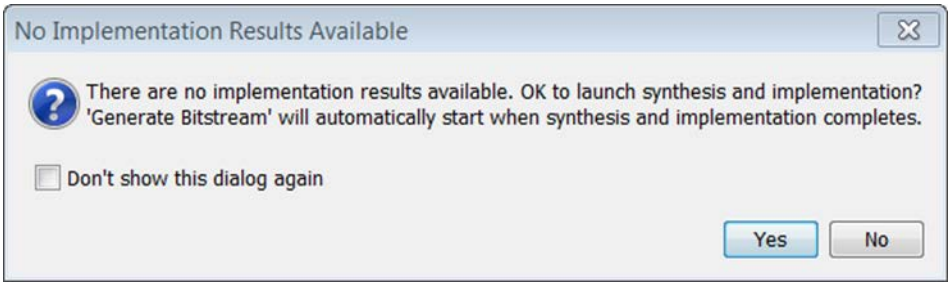

No Implementation Results Available
There are no implementation results available. OK to launch synthesis and implementation?
'Generate Bitstream' will automatically start when synthesis and implementation completes.
Don't show this dialog again
Yes
No

Figure 104: No Implementation Results Available dialog box

2. When the bitstream generation is complete, the **Bitstream Generation Completed** dialog box appears. Select **Open Hardware Manager**. Click **OK**.

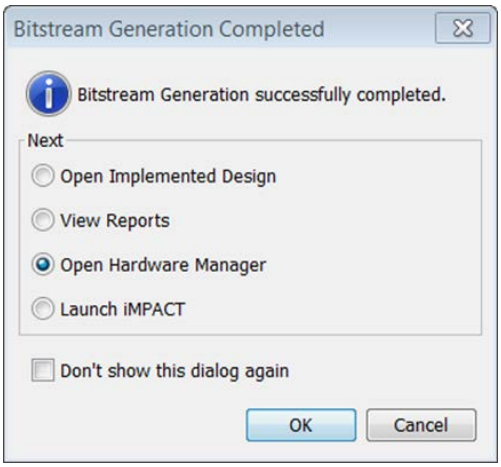


Figure 107: Bitstream Generation Completed dialog box

3. The **Hardware Manager** window appears as shown in the following figure.

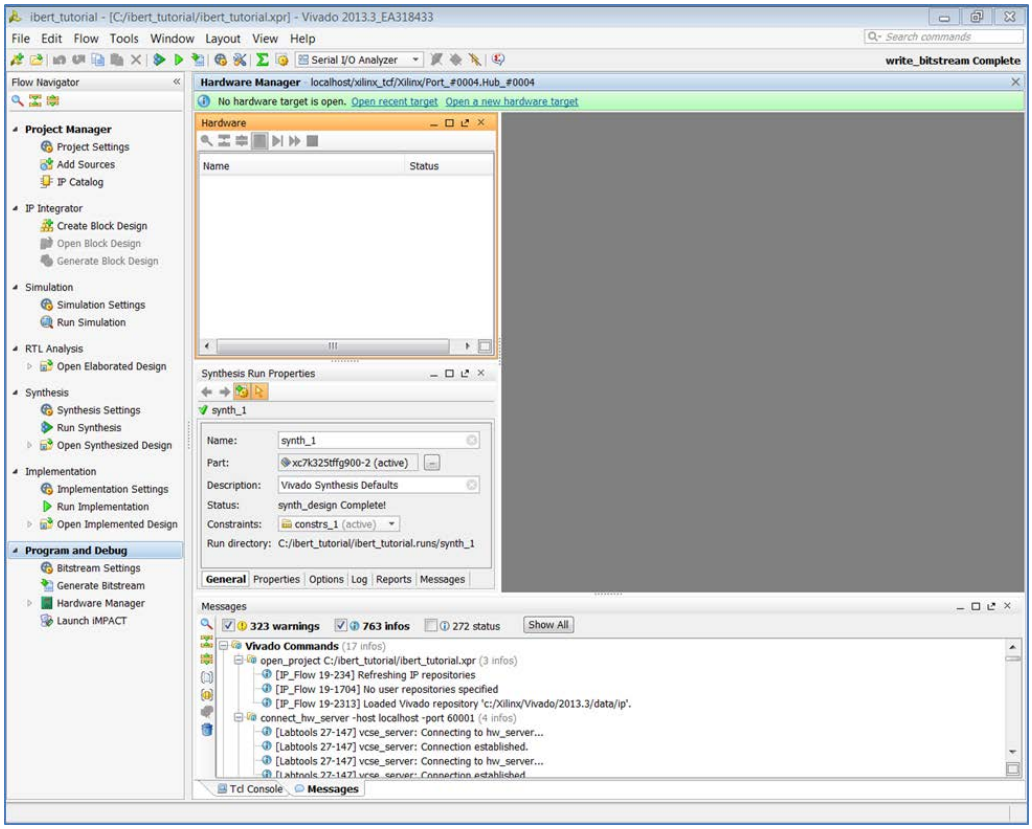


Figure 108: Hardware Manager Window

Step 4: Interact with the IBERT core using Serial I/O Analyzer

In this tutorial step, you connect to the KC705 target board, program the bitstream created in the previous step, and use the Serial I/O Analyzer to interact with the IBERT design that you created in Step 1. You perform some analysis using various input patterns and loopback modes, while observing the bit error count.

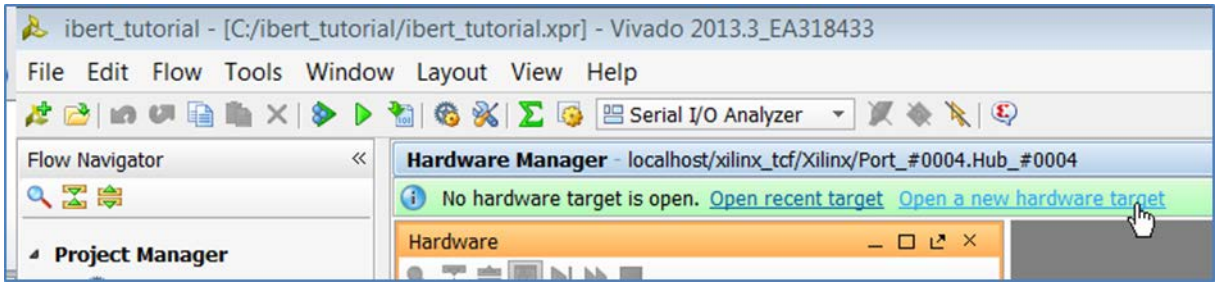

ibert_tutorial - [C:/ibert_tutorial/ibert_tutorial.xpr] - Vivado 2013.3_EA318433
File Edit Flow Tools Window Layout View Help
Serial I/O Analyzer
Flow Navigator
Project Manager
Hardware Manager - localhost/xilinx_tcf/Xilinx/Port_#0004.Hub_#0004
No hardware target is open. Open recent target Open a new hardware target
Hardware

Figure 109: Open a new hardware target

1. Click **Open a new hardware target**. When the **Open New Hardware Target** wizard opens, click **Next**.

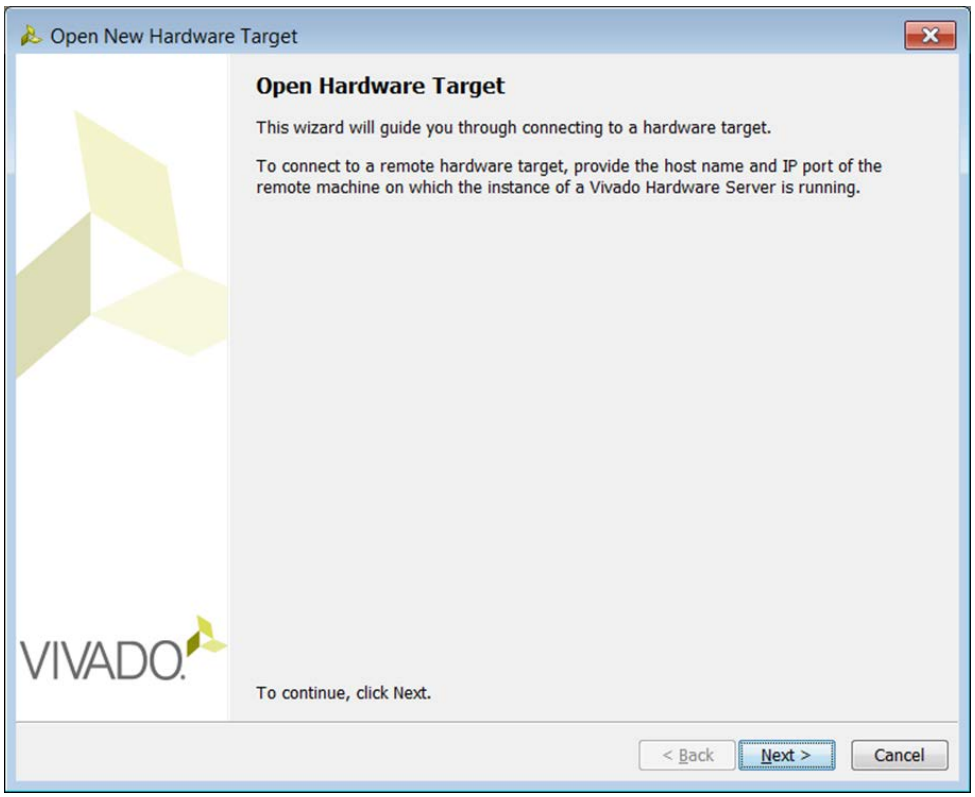

Open New Hardware Target
Open Hardware Target
This wizard will guide you through connecting to a hardware target.
To connect to a remote hardware target, provide the host name and IP port of the remote machine on which the instance of a Vivado Hardware Server is running.
VIVADO
To continue, click Next.
< Back
Next >
Cancel

Figure 110: Open New Hardware Target wizard

2. In the **Connect to** field, choose **Local server**. Click **Next**.

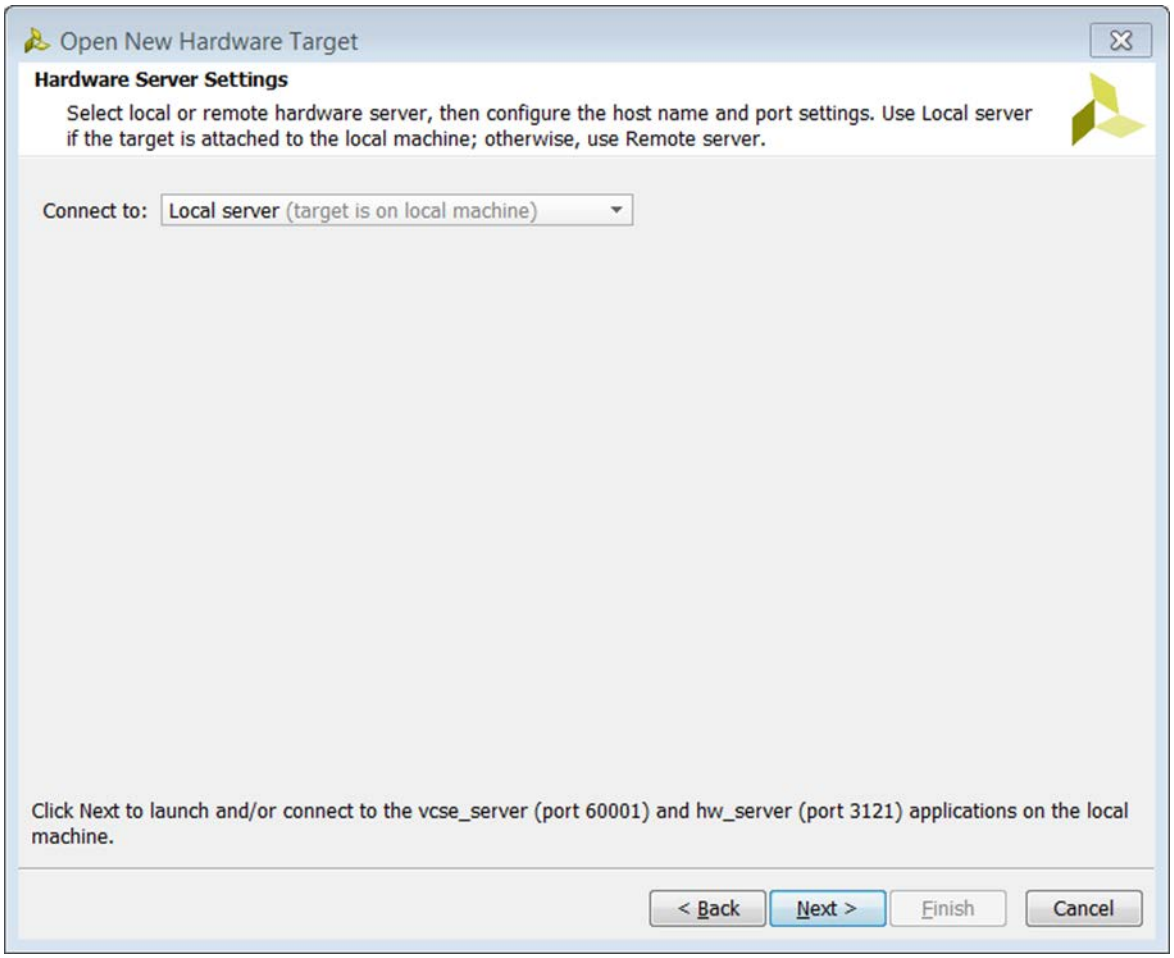

Open New Hardware Target
Hardware Server Settings
Select local or remote hardware server, then configure the host name and port settings. Use Local server if the target is attached to the local machine; otherwise, use Remote server.
Connect to:
Local server (target is on local machine)
Click Next to launch and/or connect to the vcse_server (port 60001) and hw_server (port 3121) applications on the local machine.
< Back
Next >
Finish
Cancel

Figure 111: Vivado CSE Server Name page

3. In the **Select Hardware Target** page, click **Next**. There is only one target board in this case to connect to, so the default is selected.

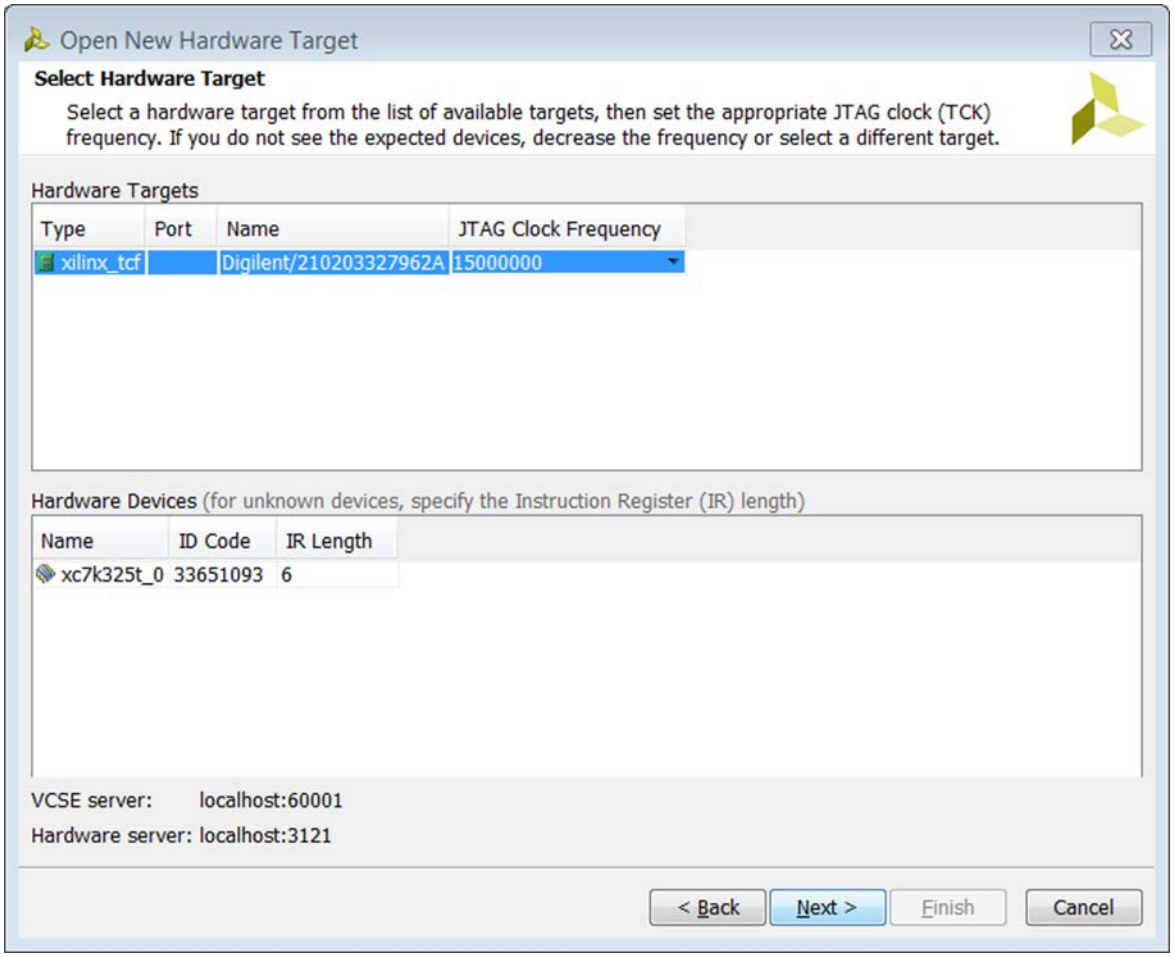


Figure 112: Select Hardware Target page

4. In the **Open Hardware Target Summary** page, review the options that you selected. Click **Finish**.

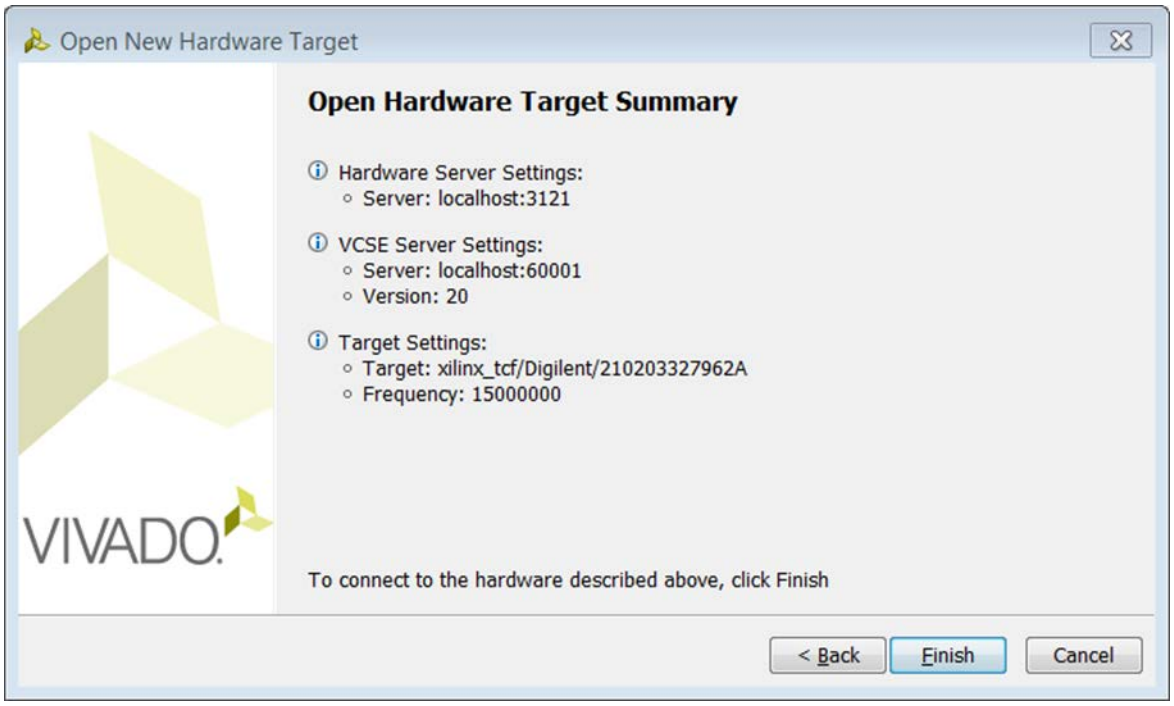


Figure 114: Open Hardware Target Summary dialog box

5. The **Hardware** window in Vivado IDE should show the status of the target FPGA device on the KC705 board.

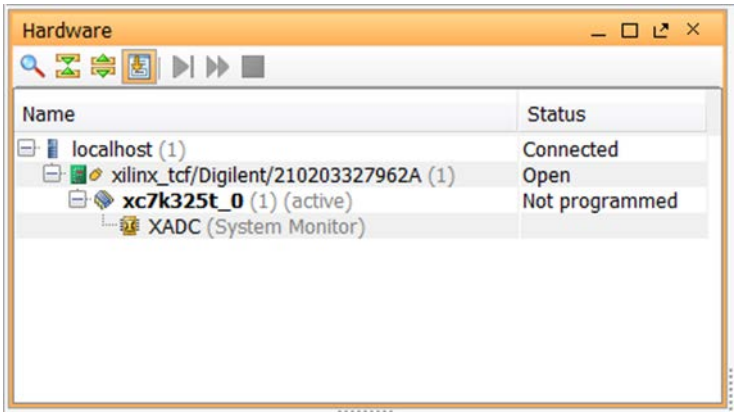


Figure 115: Hardware Window showing the XC7K325T device on the KC705 board

6. Select **XC7K325T_0(0)** in the **Hardware** window, right-click and select **Program Device**.

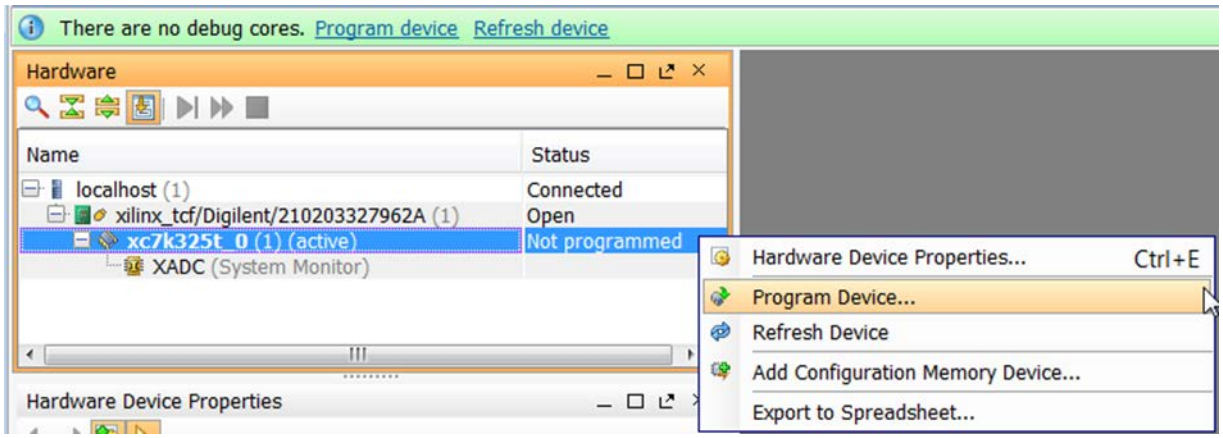


Figure 116: Program target device

7. The **Program Device** dialog box opens. Make sure that the correct bitfile is selected and click **OK**.

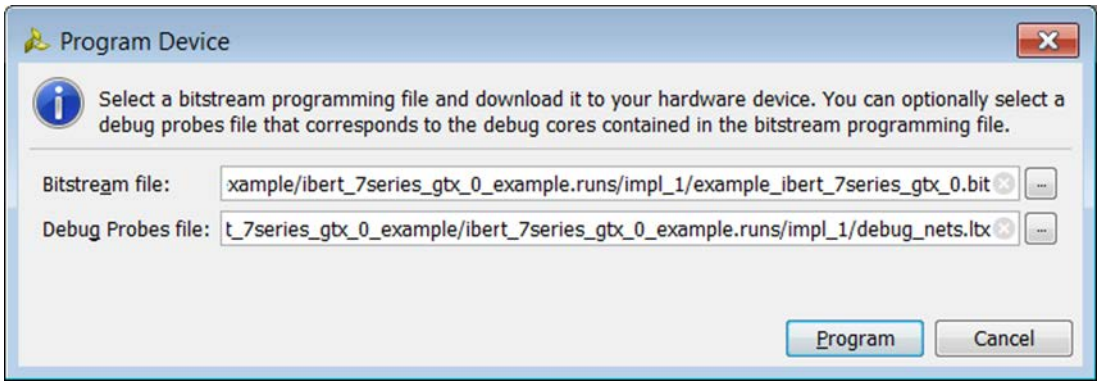

Program Device
Select a bitstream programming file and download it to your hardware device. You can optionally select a debug probes file that corresponds to the debug cores contained in the bitstream programming file.
Bitstream file:
xample/ibert_7series_gtx_0_example.runs/impl_1/example_ibert_7series_gtx_0.bit
Debug Probes file:
t_7series_gtx_0_example/ibert_7series_gtx_0_example.runs/impl_1/debug_nets.ltx
Program
Cancel

Figure 117: Program Device dialog box

8. The **Hardware** window now shows the IBERT IP that you customized and implemented from the previous steps. It contains two QUADS each of which has four GTX transceivers. These components of the IBERT were detected while scanning the FPGA after downloading the bitstream. If you do not see the QUADS then select the **XC7K325** device, right-click and select **Refresh Device**.

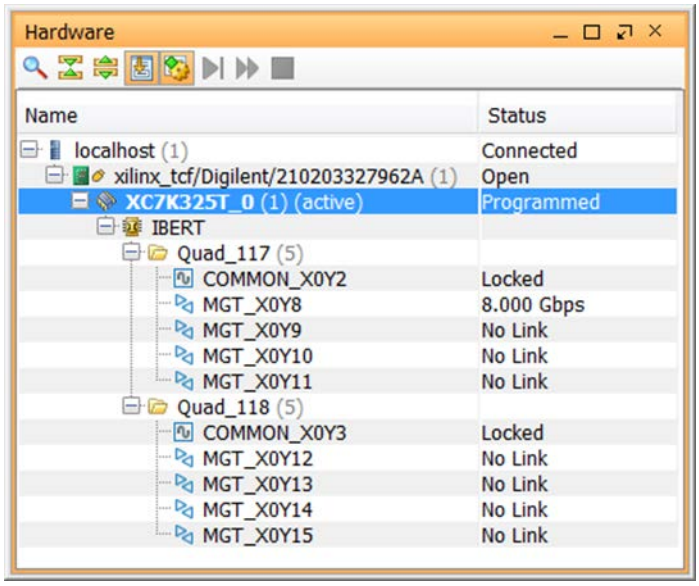


Figure 118: The Hardware window showing the QUADS after the FPGA has been programmed

9. Next, create links for all eight transceivers. Vivado Serial I/O analyzer is a link-based analyzer, which allows users to link between any transmitter and receiver GTs within the IBERT design. For this tutorial, simply link the TX and RX of the same channel. To create a link, right-click the **IBERT Core** in the **Hardware** window and click **Create Links**.

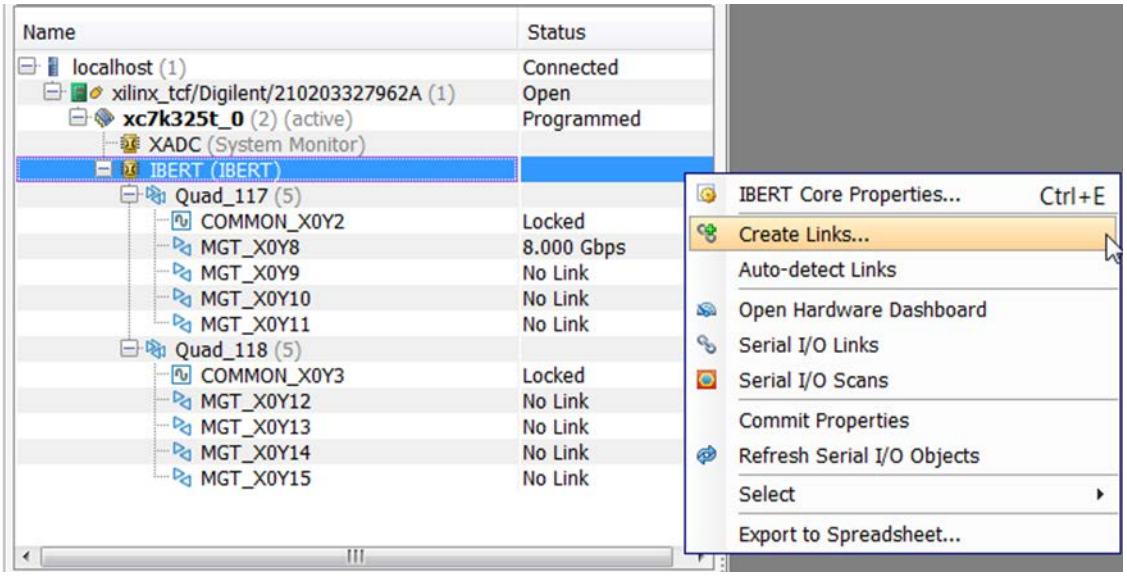


Figure 119: Create Links

The **Create Links** dialog box opens. Make sure the first transceiver pairs (MGT_X0Y8/TX and MGT_X0Y8/RX) are selected.

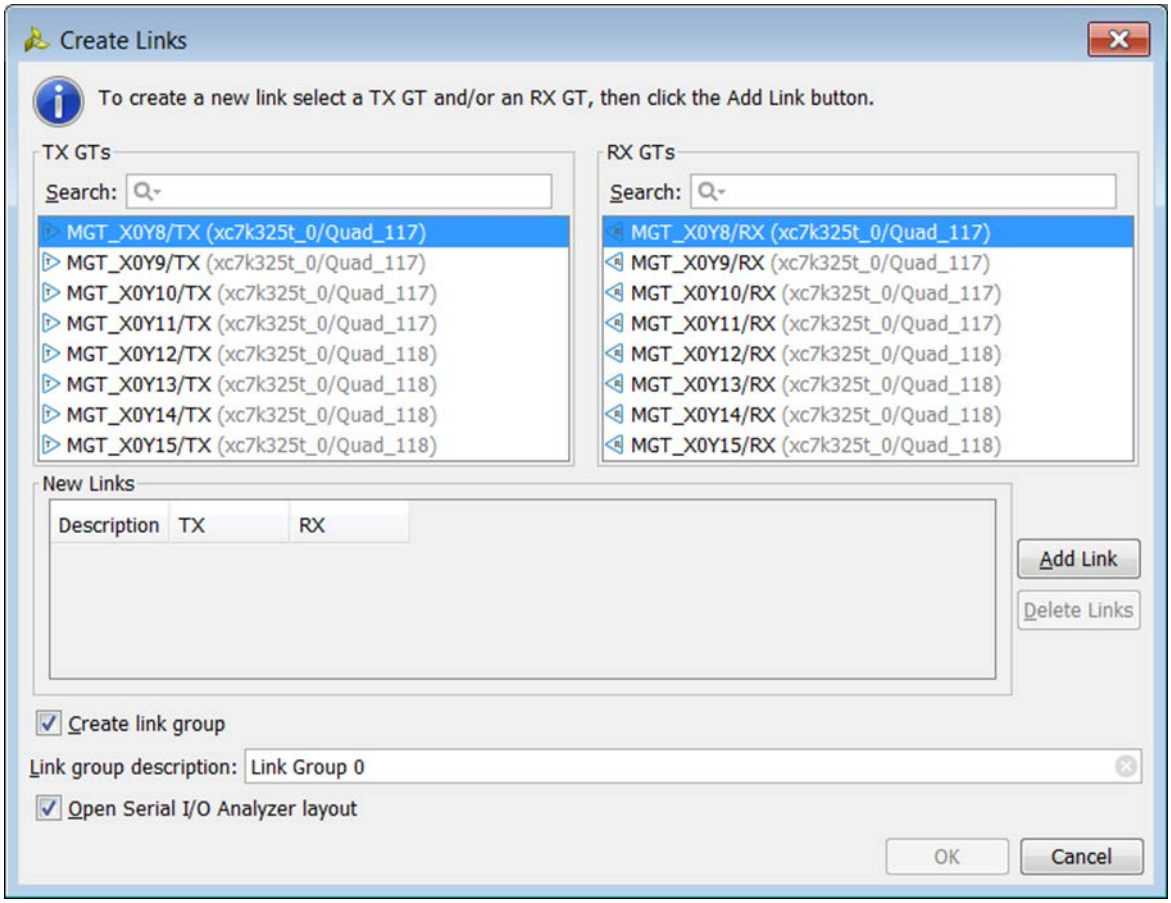


Figure 120: Selecting the transceiver pairs for creating new links

Click **Add Link**. In the **Link group description** field, type **Link Group SMA**. Click **OK**.

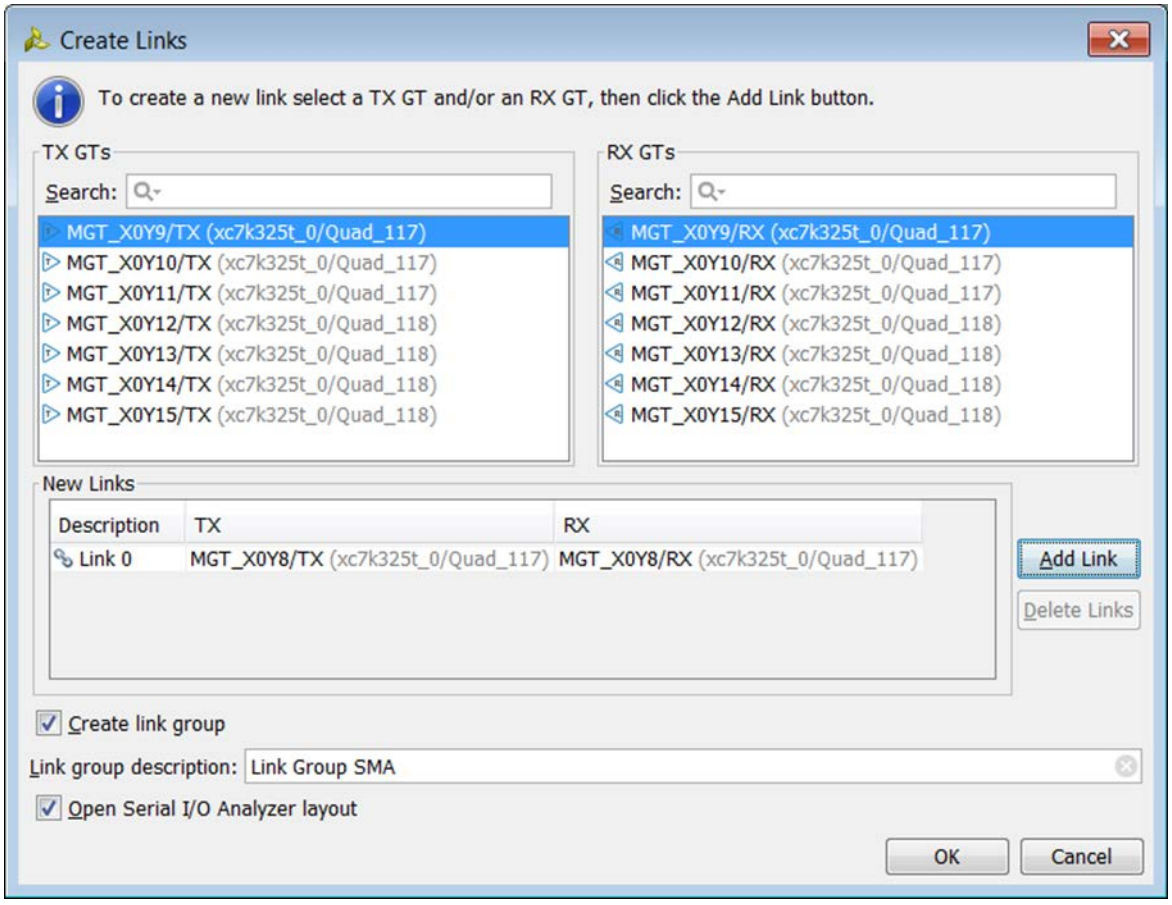


Figure 121: Create Links dialog box

For the first link group, call this Link Group SMA as this is the only transceiver channel that is linked through the SMA cables. The new link shows up in the **Links** window.

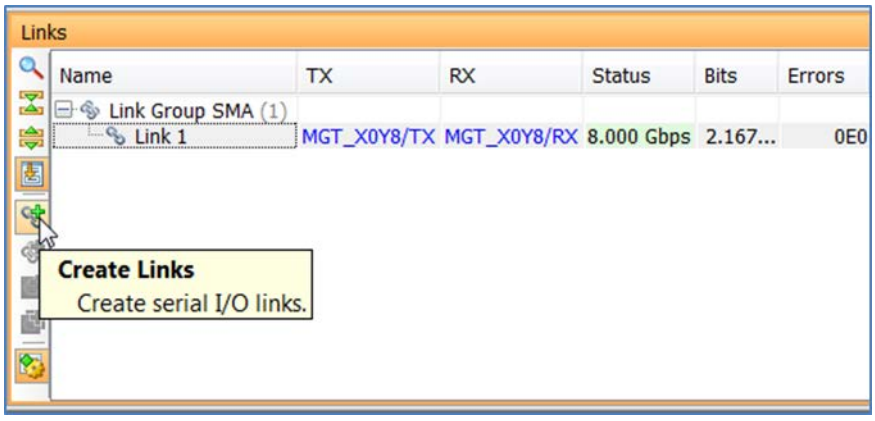


Figure 122: Create link groups for other transceiver pairs

Click **Create Link** again to create link groups for the rest of the transceiver pairs. To do this ensure that the transceiver pairs are selected, and click the **+** sign icon (add new link) repeatedly, until all the links have been added to the new link group called **Link Group Internal Loopback**. Click **OK**.

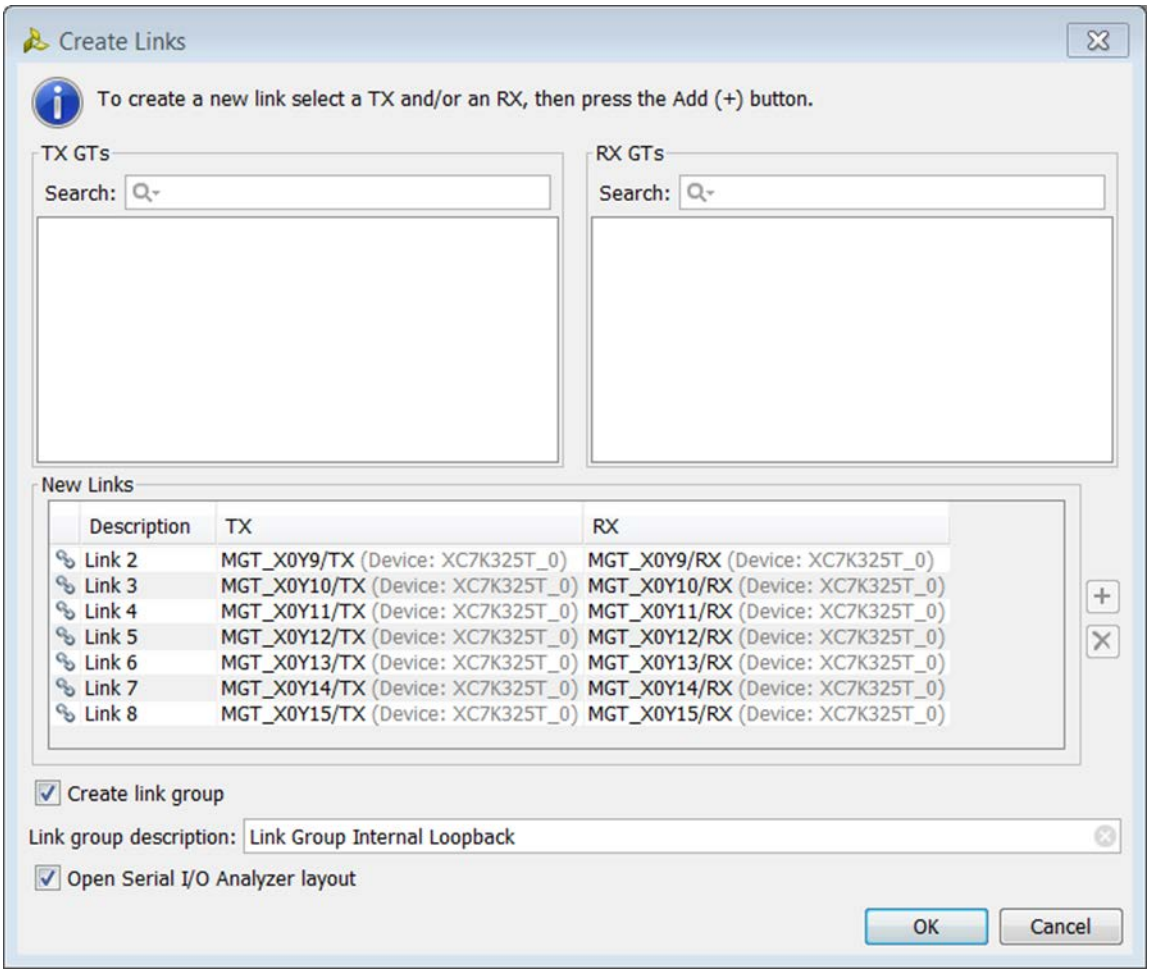

Create Links
To create a new link select a TX and/or an RX, then press the Add (+) button.
TX GTs
Search:
RX GTs
Search:
New Links
Description
TX
RX
Link 2 MGT_X0Y9/TX (Device: XC7K325T_0) MGT_X0Y9/RX (Device: XC7K325T_0)
Link 3 MGT_X0Y10/TX (Device: XC7K325T_0) MGT_X0Y10/RX (Device: XC7K325T_0)
Link 4 MGT_X0Y11/TX (Device: XC7K325T_0) MGT_X0Y11/RX (Device: XC7K325T_0)
Link 5 MGT_X0Y12/TX (Device: XC7K325T_0) MGT_X0Y12/RX (Device: XC7K325T_0)
Link 6 MGT_X0Y13/TX (Device: XC7K325T_0) MGT_X0Y13/RX (Device: XC7K325T_0)
Link 7 MGT_X0Y14/TX (Device: XC7K325T_0) MGT_X0Y14/RX (Device: XC7K325T_0)
Link 8 MGT_X0Y15/TX (Device: XC7K325T_0) MGT_X0Y15/RX (Device: XC7K325T_0)
Create link group
Link group description: Link Group Internal Loopback
Open Serial I/O Analyzer layout
OK
Cancel

Figure 123: Create Link dialog box to create the second link group

10. Once the links have been created, they are added to the **Links** window as shown.

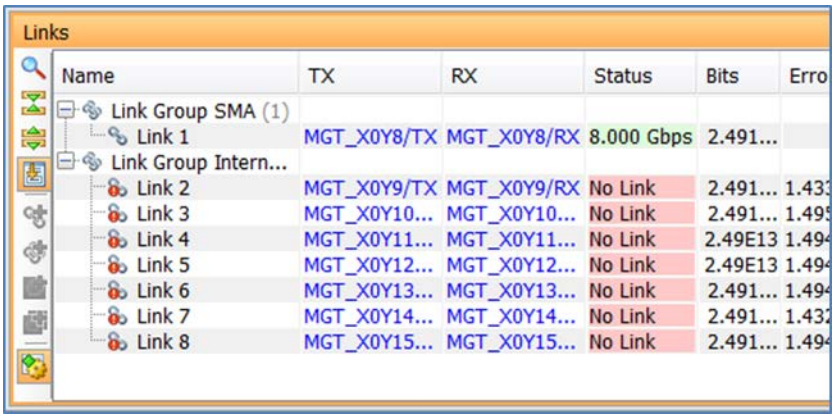


Figure 124: Links window after all the link groups have been created

As expected, only **Link1** indicates a link with **8.0 Gbps** line rate and the rest of the GTX channels show **No Link** status.

For more information about the different columns of the **Links** windows, refer to Vivado User Guide: Programming and Debugging, (UG908).

11. You can fix the **No Link** status for all the links by changing the GT properties of these channels to one of the loopback modes. To do this, select the appropriate transceiver in the **Hardware** window and modify the GT property in the **GT Properties** window.

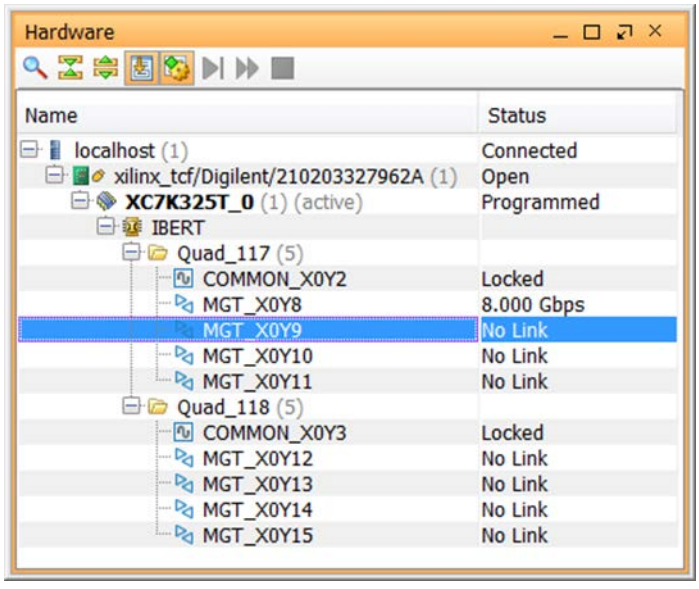


Figure 125: Select the transceiver whose property needs to be modified

In the **GT Properties** window, under the **Properties** tab, select **LOOPBACK** pull-down menu and select **Near-End PCS**.

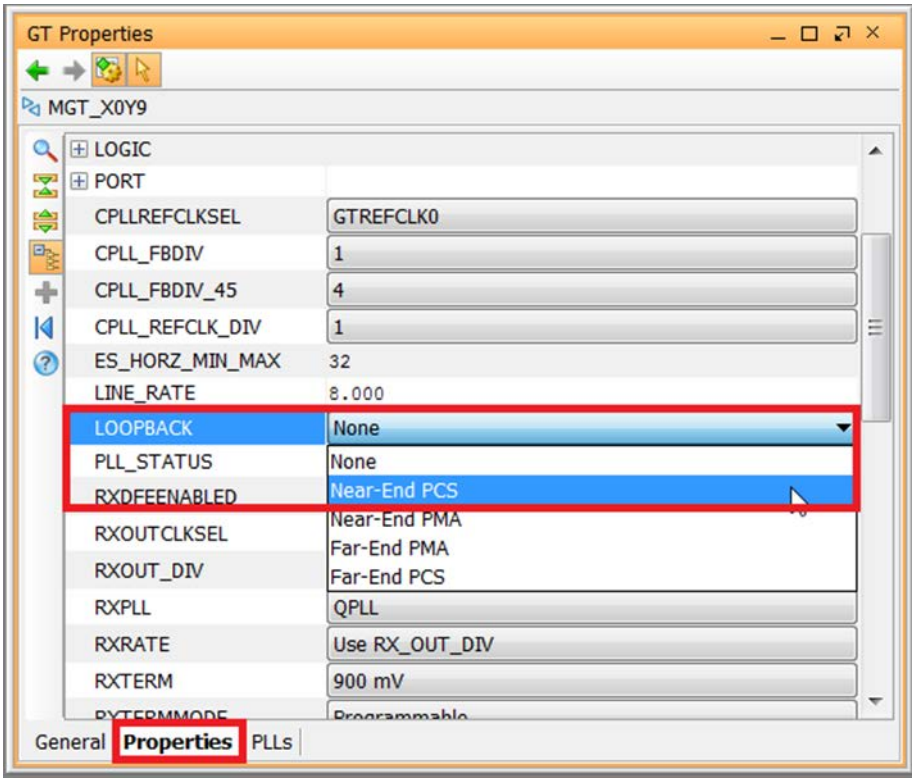


Figure 126: Selecting appropriate Loopback mode for transceivers

This changes the **Status** in the **Link** window for the transceiver in question as shown.

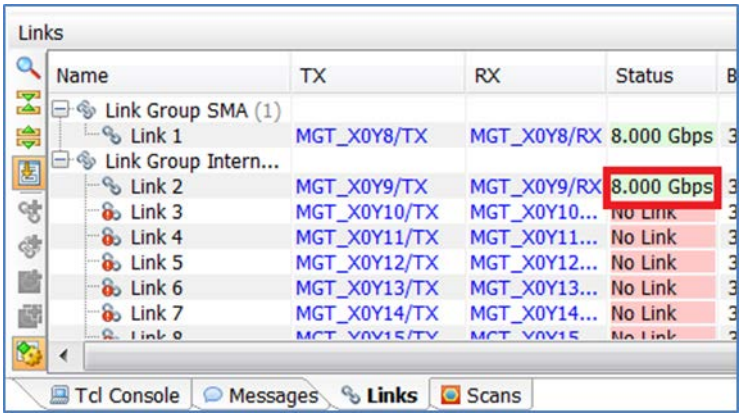


Figure 127: Link Status after changing GT Properties to Near-end PCS

Change the GT properties of the rest of the transceivers as described above.

12. Next, create a 2D scan. Click **Create Scan** in the **Links** window.

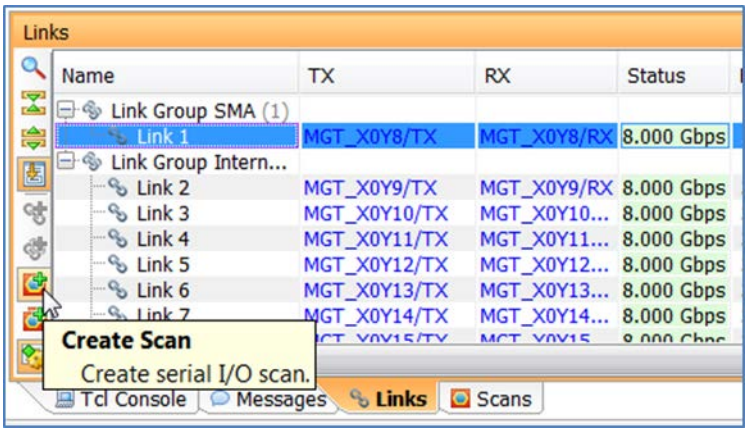


Figure 128: Creating a 2D Scan for Link 1

The **Create Scan** dialog box opens. In this dialog box, you can change the various scan properties. In this case, leave everything to its default value and click **OK**. For more information on the scan properties, see Vivado User Guide: Programming and Debugging, (UG908).

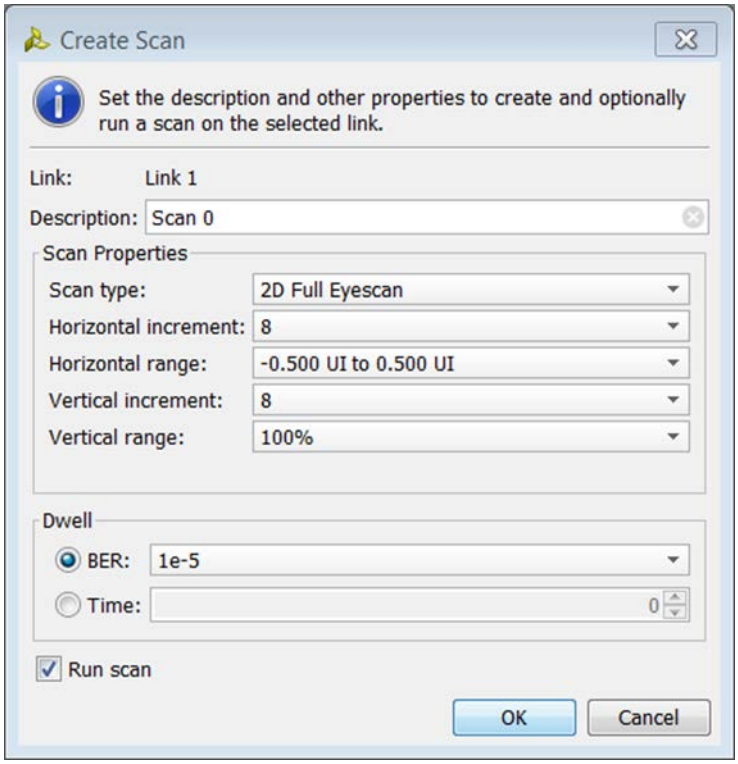


Figure 129: The Create Scan dialog box

The **Scan Plot** window opens as shown below.

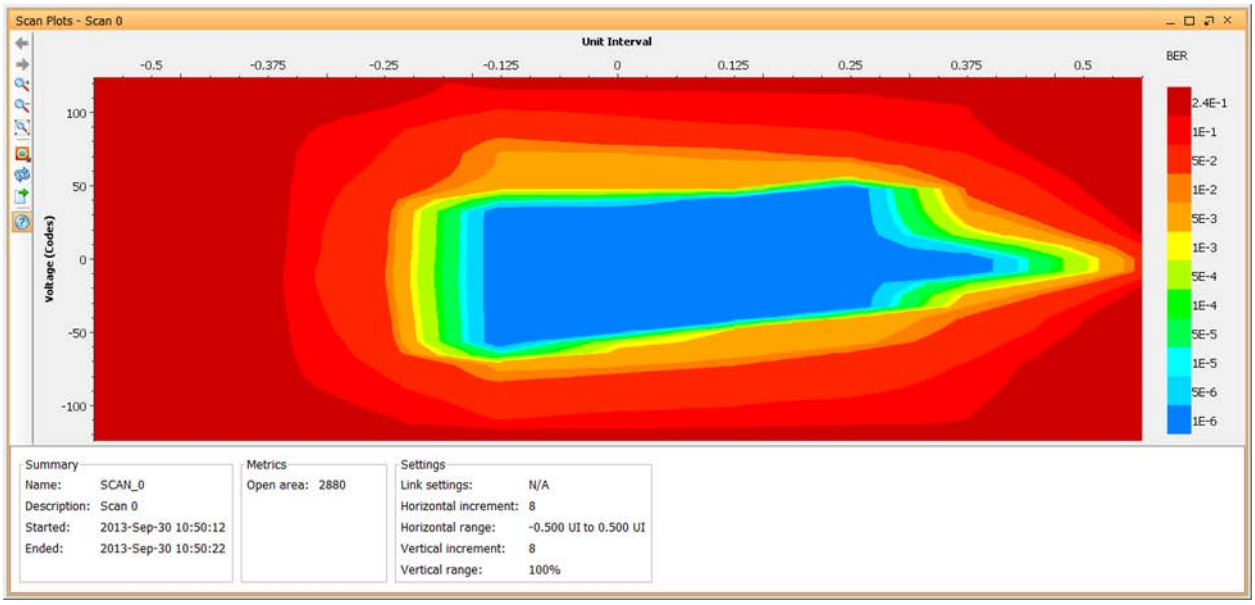


Figure 130: 2D Scan Plot

The 2D Scan Plot is a heat map of the BER value.

13. You can also perform a Sweep test on the links that you created earlier. In the **Links** window, highlight **Link 0** under the Link called Link **Group SMA**, right-click and select **Create Sweep**.

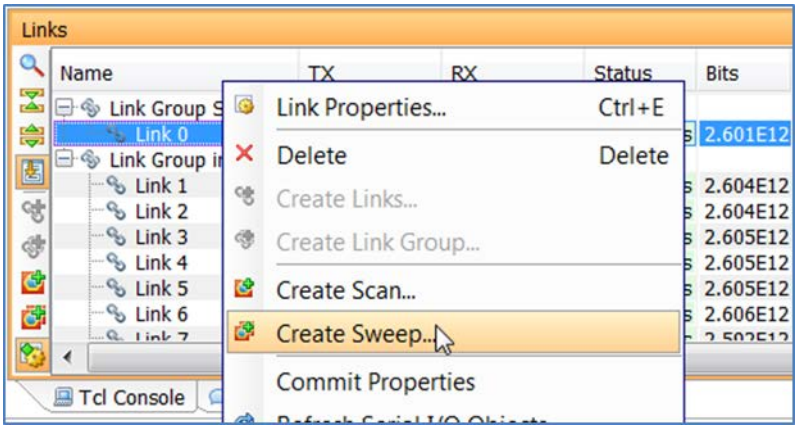


Figure 131: Create a Sweep Test

14. The **Create Sweep** dialog box opens, as shown below. Various properties for the Sweep test can be changed in this dialog box. Leave all the values to its default state and click **OK**.

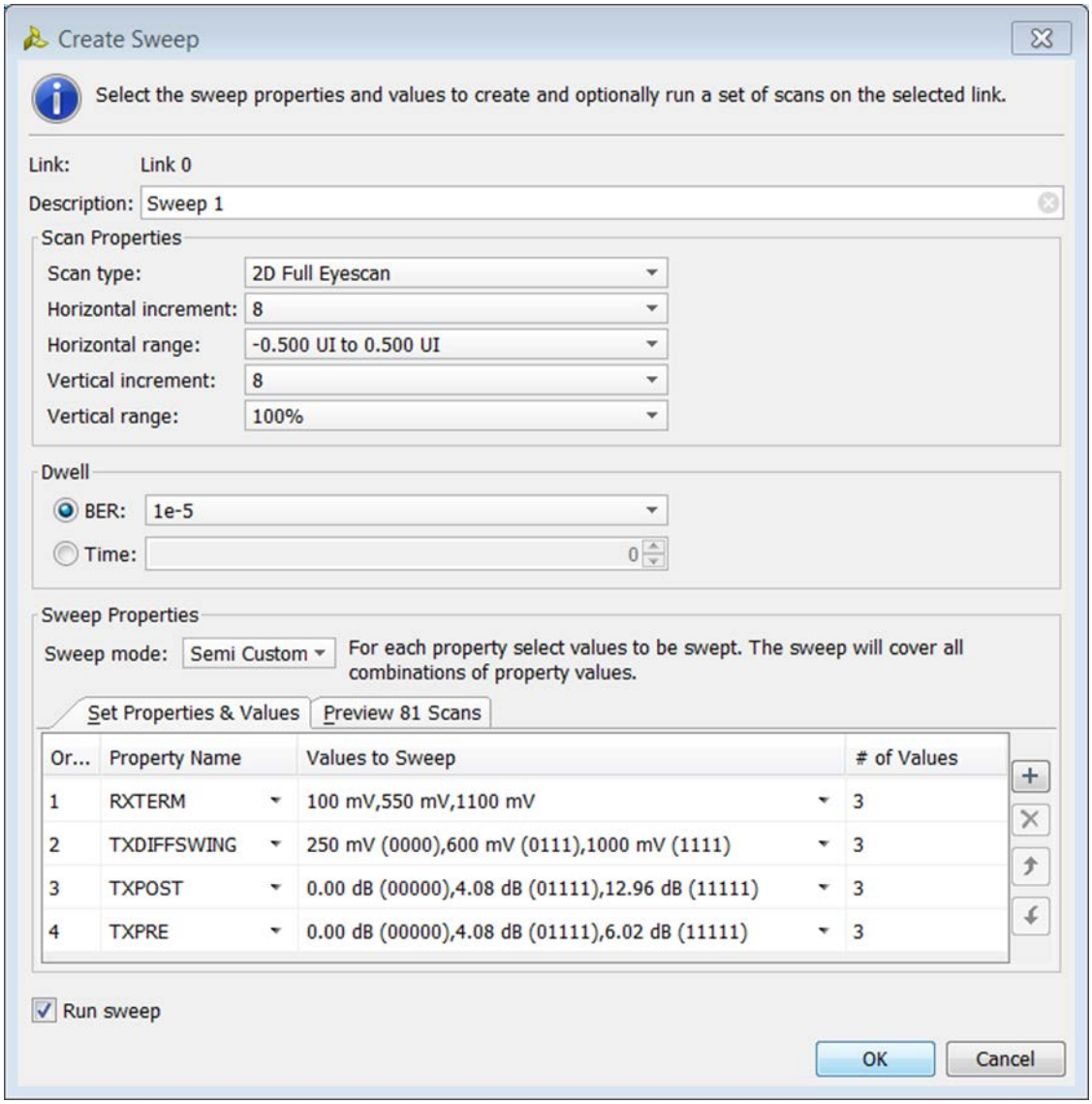

Create Sweep
Select the sweep properties and values to create and optionally run a set of scans on the selected link.
Link: Link 0
Description: Sweep 1
Scan Properties
Scan type: 2D Full Eyescan
Horizontal increment: 8
Horizontal range: -0.500 UI to 0.500 UI
Vertical increment: 8
Vertical range: 100%
Dwell
BER: 1e-5
Time: 0
Sweep Properties
Sweep mode: Semi Custom
For each property select values to be swept. The sweep will cover all combinations of property values.
Set Properties & Values
Preview 81 Scans
Or... Property Name Values to Sweep # of Values
1 RXTERM 100 mV,550 mV,1100 mV 3
2 TXDIFFSWING 250 mV (0000),600 mV (0111),1000 mV (1111) 3
3 TXPOST 0.00 dB (00000),4.08 dB (01111),12.96 dB (11111) 3
4 TXPRE 0.00 dB (00000),4.08 dB (01111),6.02 dB (11111) 3
Run sweep
OK
Cancel

Figure 132: Create Sweep dialog box

Since there are four different Sweep Properties and each of these properties has three different values (as seen in the **Values to Sweep** column), a total number of 81 sweep tests are carried out. The **Scans** window shows the results of all the scans that have been done for the selected link.

CAUTION! *Since there are 81 scans to be done, it could be a few minutes before all the scans are complete.*

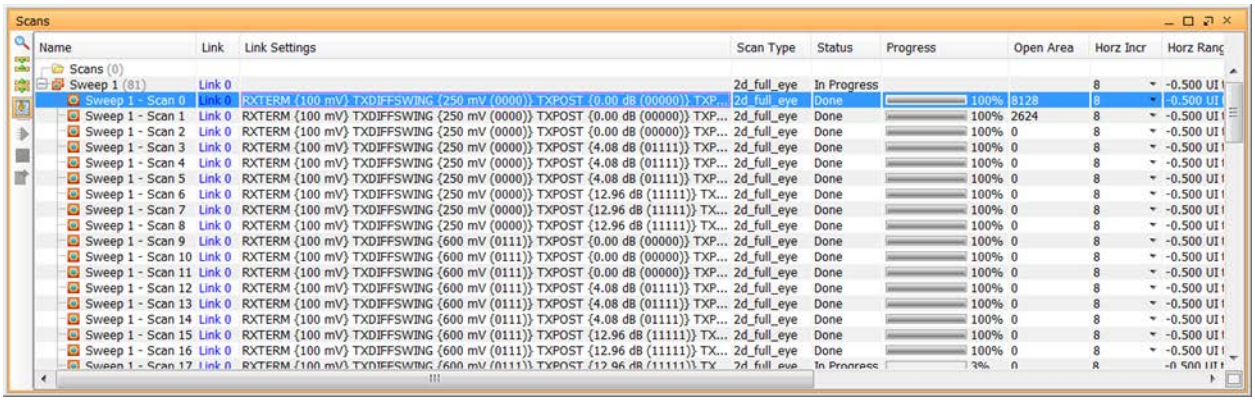


Figure 133: Sweep Test results in the Scans window

To see the results of any of the scans that have been performed, highlight the scan, right-click, and select **Display Scan Plots**.

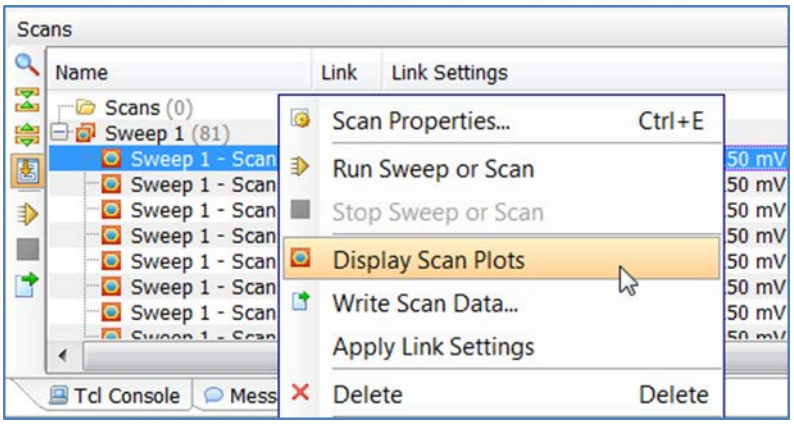


Figure 134: Displaying Scan Plots

The **Scan Plots** window opens showing the details of the scan performed.

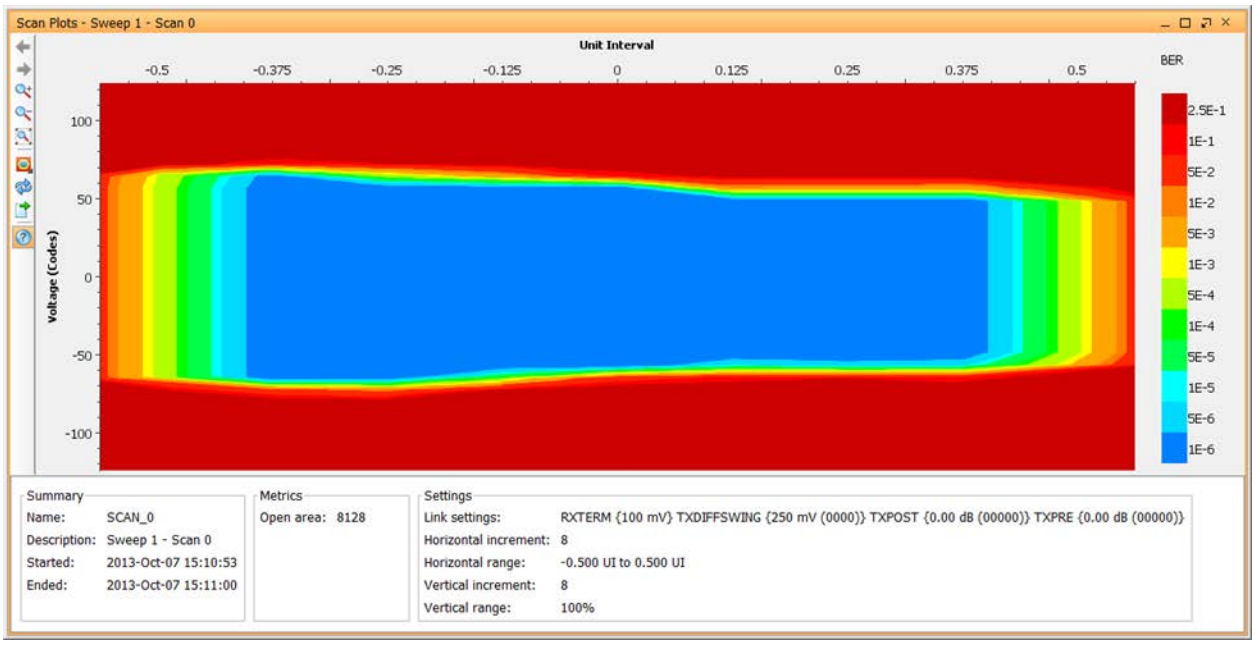


Figure 135: Analyzing the results of individual scans

Lab 6: Using Vivado ILA core to Debug JTAG-AXI Transactions

Introduction

The purpose of this tutorial is to provide a very quick and easy to reproduce introduction to inserting an ILA core into the JTAG to AXI Master IP core example design, and using the ILA's advanced trigger and capture capabilities.

What is the JTAG to AXI Master IP core?

The LogiCORE™ IP JTAG-AXI core is a customizable core that can generate AXI transactions and drive AXI signals internal to FPGA at run-time. This supports all memory-mapped AXI interfaces (except AXI4-Stream) and Lite protocol and can be selected using a parameter. The width of AXI data bus is customizable. This IP can drive any AXI4-Lite or Memory Mapped Slave directly. This can also be connected as master to the interconnect. Run-time interaction with this core requires the use of the Vivado® logic analyzer feature.

Key Features

- AXI4 master interface
- Option to select AXI4-Memory Mapped and AXI4-Lite interfaces
- User controllable AXI read and write enable
- User Selectable AXI datawidth : 32 and 64
- User Selectable AXI ID width up to four bits
- Vivado logic analyzer Tcl Console interface to interact with hardware

Additional Documentation

LogiCORE IP JTAG AXI Master v1.0 Product Guide (AXI), (PG174) contains more information the JTAG to AXI Master IP core.

Design Description

This section has three steps as follows:

1. Opening the JTAG to AXI Master IP Example Design project and adding mark_debug to the AXI interface connection. Inserting an ILA 3.0 core into the design and configuring it for advanced trigger is also included in this step.
2. Programming the KC705 board and interacting with the JTAG to AXI Master IP core.
3. Using the ILA 3.0 Advanced Trigger Feature to Trigger on an AXI Read Transaction.

Step 1: Opening the JTAG to AXI Master IP Example Design and Configuring the AXI Interface Debug Connections

To create a project, use the New Project wizard to name the project, add RTL source files and constraints, and specify the target device.

1. Invoke the Vivado IDE.
2. In the **Getting Started** screen, click **Create New Project** to start the New Project wizard. Click **Next**.
3. In the **Project Name** page, name the new project **jtag_2_axi_tutorial** and provide the project location (`C:/jtag_2_axi_tutorial`). Ensure that **Create Project Subdirectory** is selected. Click **Next**.
4. In the **Project Type** page, specify the **Type of Project** to create as **RTL Project**. Click **Next**.
5. In the **Add Sources** page click **Next**.
6. In the **Add Existing IP** page click **Next**.
7. In the **Add Constraints** page click **Next**.

8. In the **Default Part** page choose **Boards** and choose the **Kintex-7 KC705 Evaluation Platform**. Click **Next**.

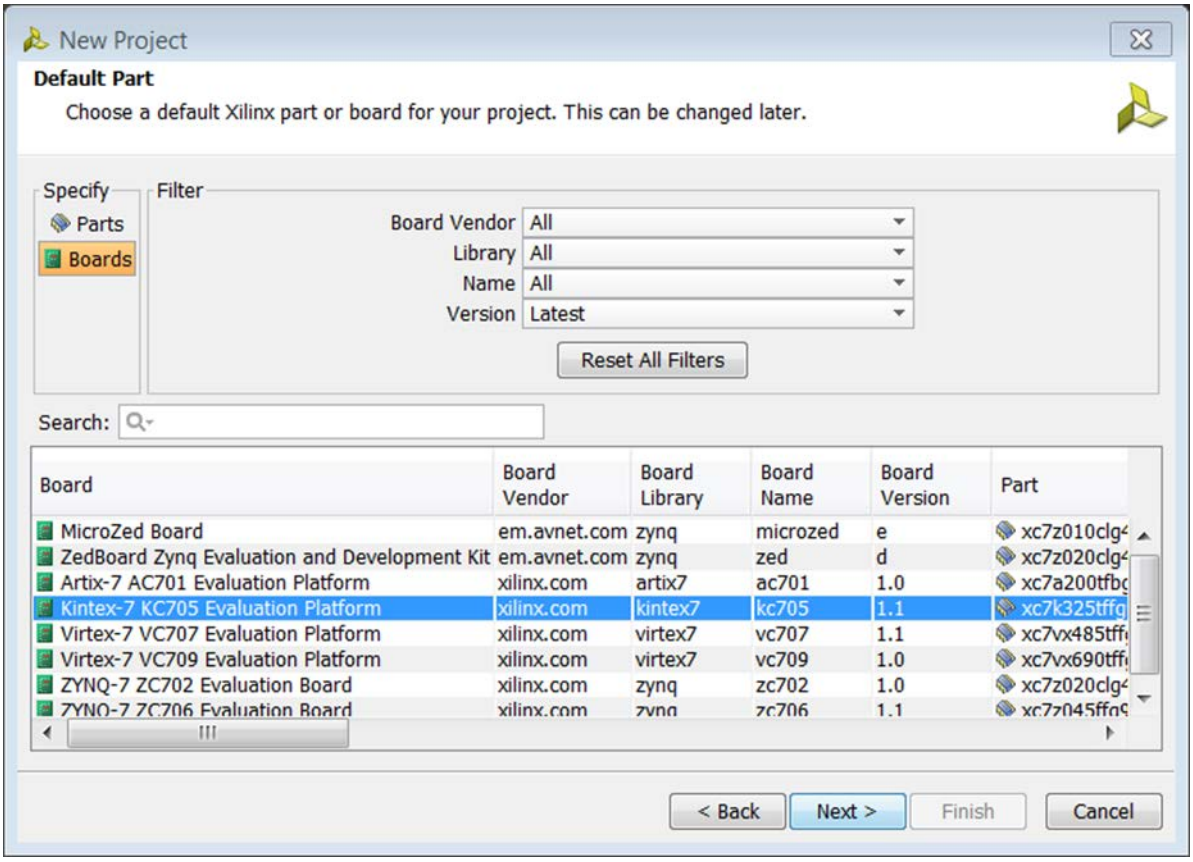

New Project
Default Part
Choose a default Xilinx part or board for your project. This can be changed later.
Specify
Parts
Boards
Filter
Board Vendor All
Library All
Name All
Version Latest
Reset All Filters
Search:
Board
Board Vendor
Board Library
Board Name
Board Version
Part
MicroZed Board em.avnet.com zynq microzed e xc7z010clg
ZedBoard Zynq Evaluation and Development Kit em.avnet.com zynq zed d xc7z020clg
Artix-7 AC701 Evaluation Platform xilinx.com artix7 ac701 1.0 xc7a200tfb
Kintex-7 KC705 Evaluation Platform xilinx.com kintex7 kc705 1.1 xc7k325tffg
Virtex-7 VC707 Evaluation Platform xilinx.com virtex7 vc707 1.1 xc7vx485tff
Virtex-7 VC709 Evaluation Platform xilinx.com virtex7 vc709 1.0 xc7vx690tff
ZYNQ-7 ZC702 Evaluation Board xilinx.com zynq zc702 1.0 xc7z020clg
< Back
Next >
Finish
Cancel

Figure 136: Choosing the Kintex-7 KC705 Evaluation Platform board

9. In the **New Project Summary** page click **Finish**.

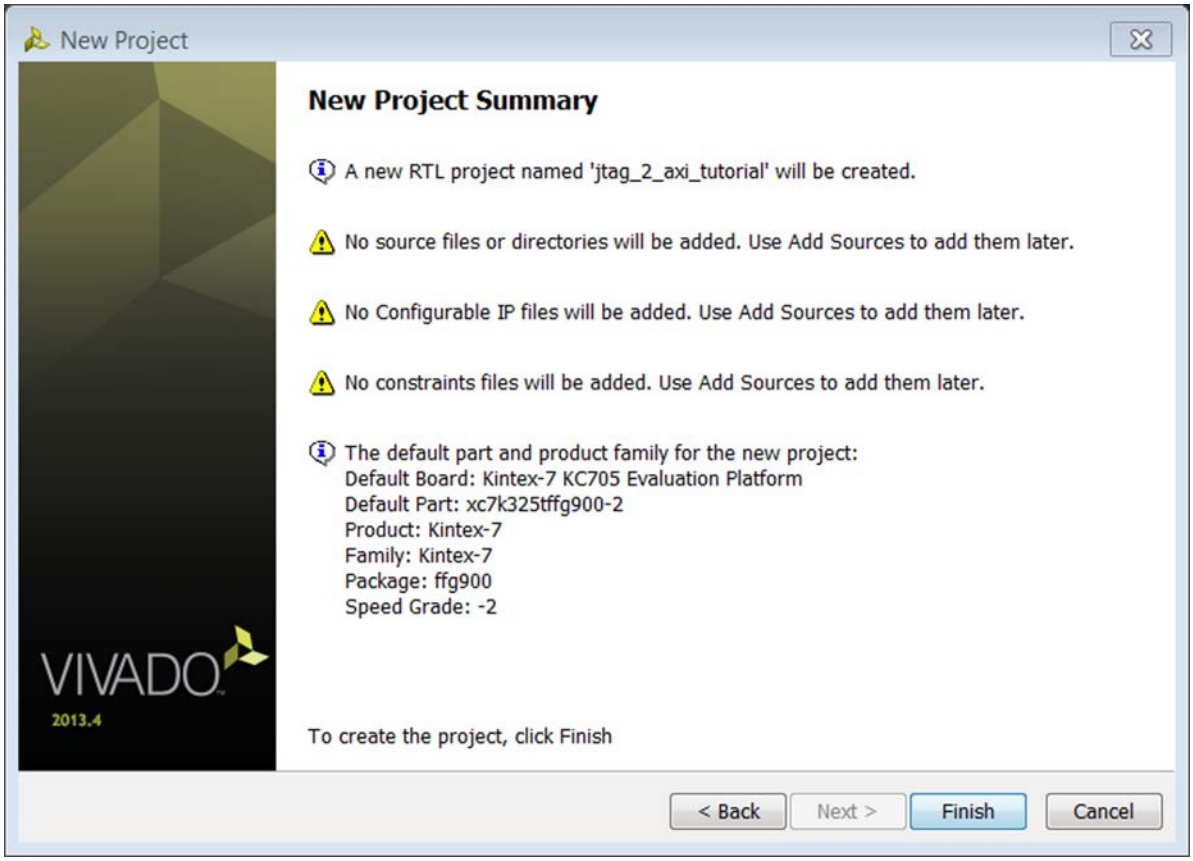
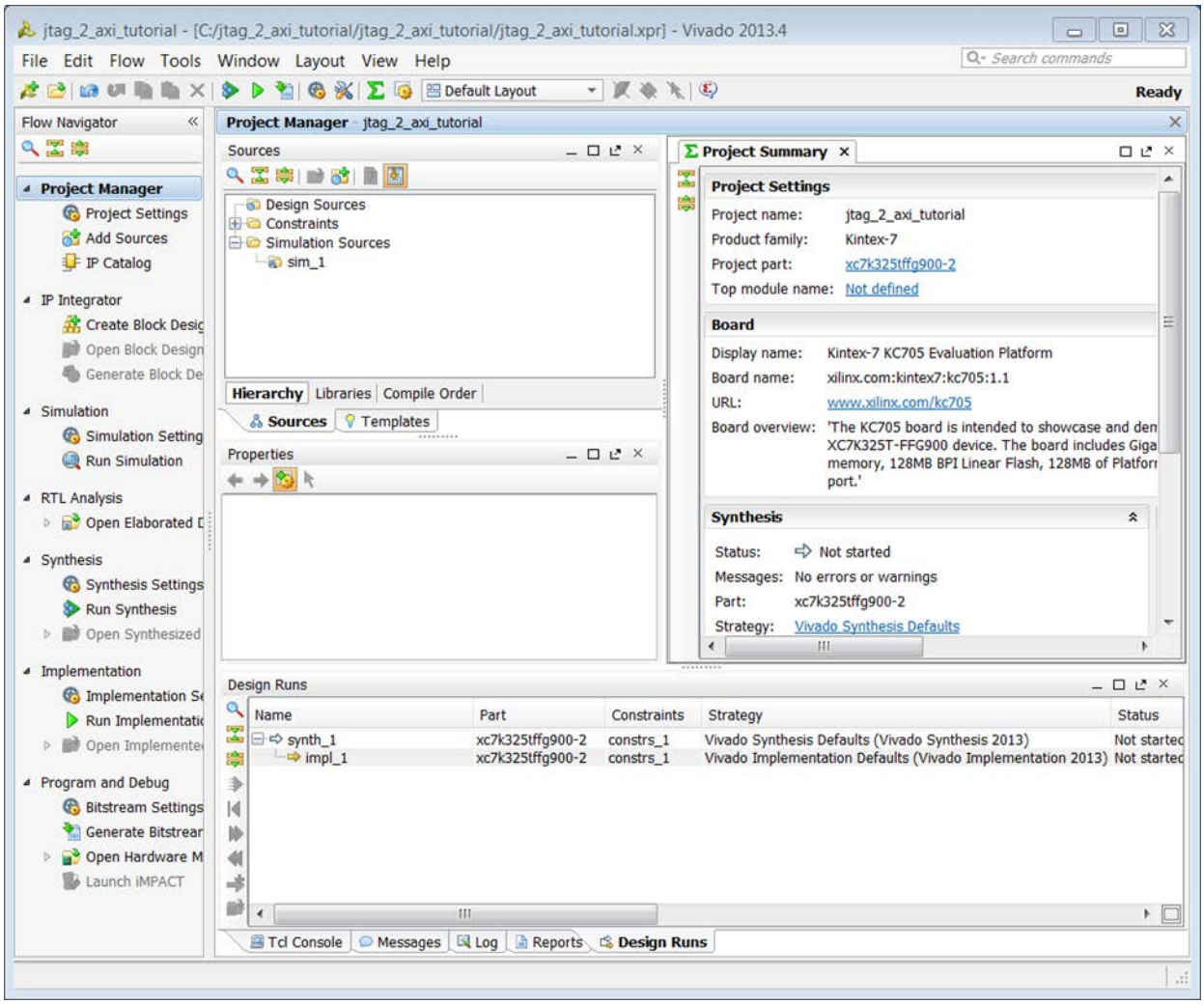

New Project
New Project Summary
A new RTL project named 'jtag_2_axi_tutorial' will be created.
No source files or directories will be added. Use Add Sources to add them later.
No Configurable IP files will be added. Use Add Sources to add them later.
No constraints files will be added. Use Add Sources to add them later.
The default part and product family for the new project:
Default Board: Kintex-7 KC705 Evaluation Platform
Default Part: xc7k325tffg900-2
Product: Kintex-7
Family: Kintex-7
Package: ffg900
Speed Grade: -2
VIVADO
2013.4
To create the project, click Finish
< Back
Next >
Finish
Cancel

Figure 137: New Project Summary

10. In the leftmost panel of the Flow Navigator under **Project Manager**, click **IP Catalog.**



11. In the **Search** field on the right of the **IP Catalog** tab, type in **JTAG to AXI**.

    ***Note**: The JTAG to AXI Master core shows up under the **Debug & Verification -> Debug** category.*

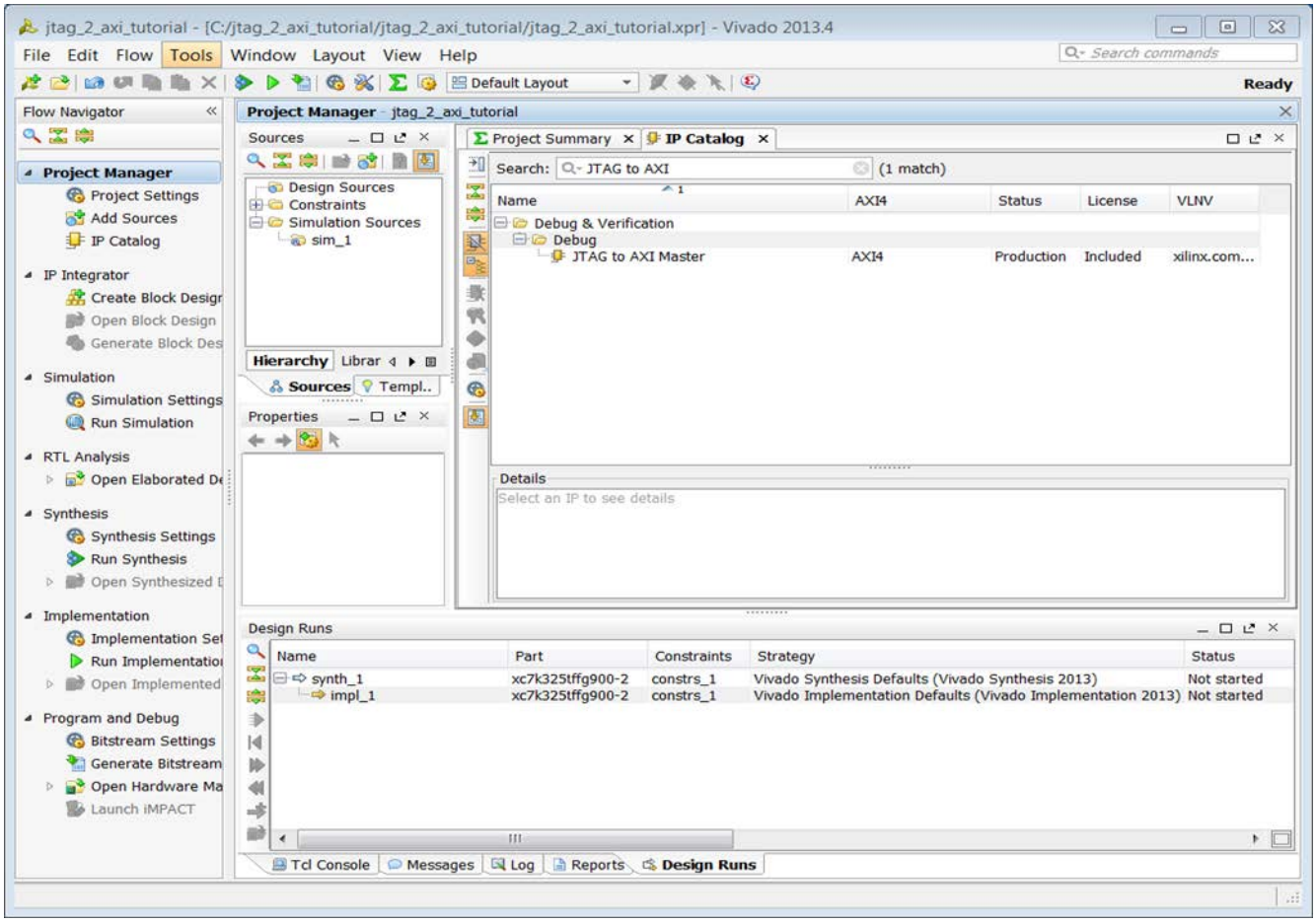


Figure 139: JTAG to AXI Master IP Core

12. Double-click **JTAG to AXI Master** core. The **Customization** dialog of the core appears. Accept the default core settings by clicking **OK**.

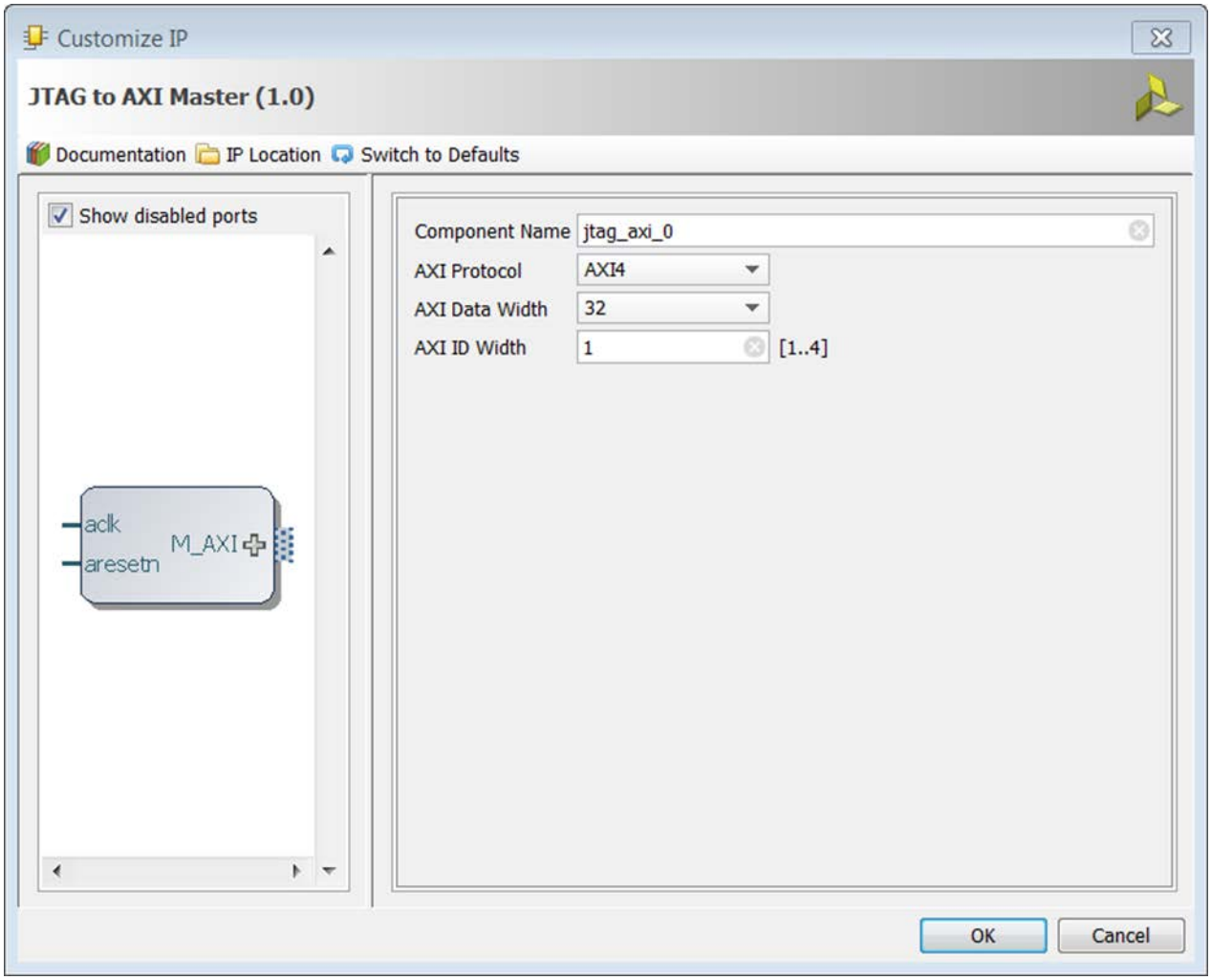


Figure 140: JTAG to AXI Master Customization Dialog

13. In the **Generate Output Products** dialog, click **Generate**.

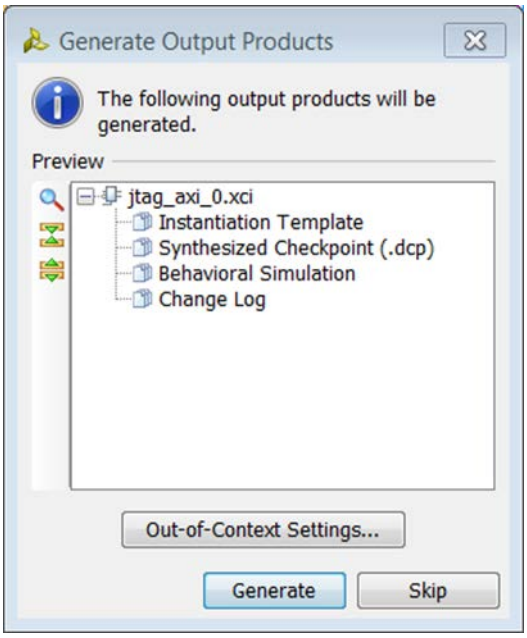


Figure 141: Generate Output Products dialog

14. The **jtag_axi_0 IP** core is inserted into the design.

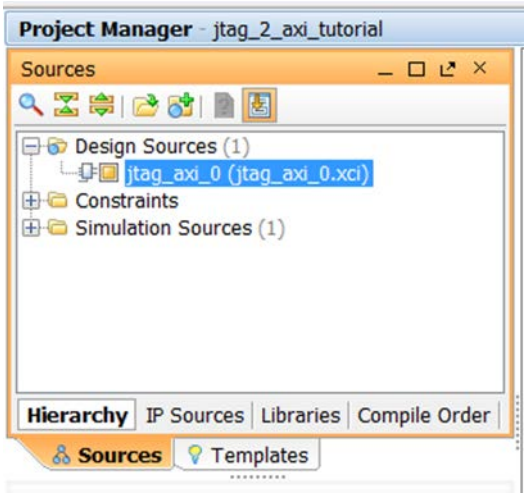


Figure 142: Generated JTAG to AXI Master IP in the design

15. Right-click **jtag_axi_0** and select **Open IP Example Design**.

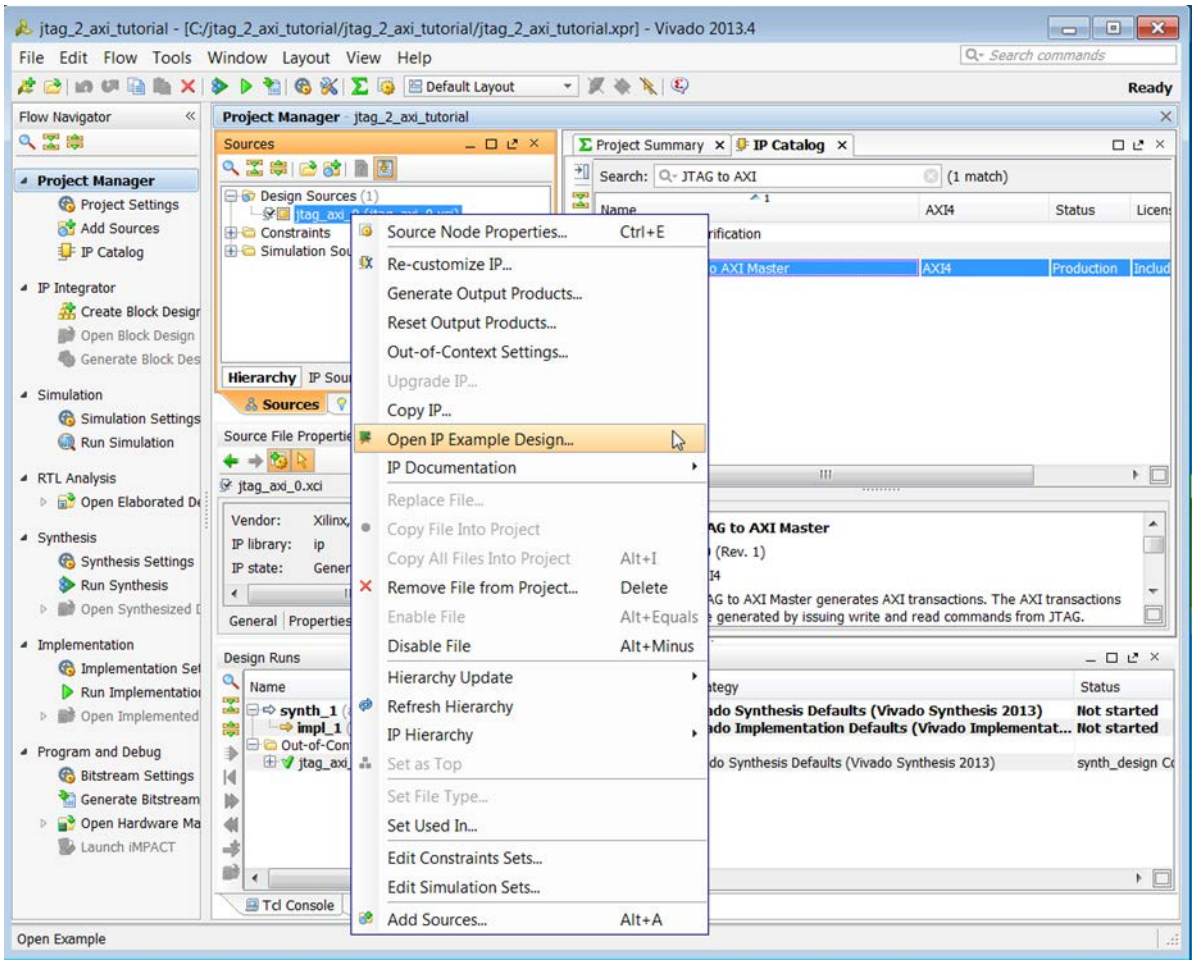


Figure 143: Open IP Example Design menu item

16. In the **Open IP Example Design** dialog, ensure that **Overwrite existing example project** is selected. Click **OK**.

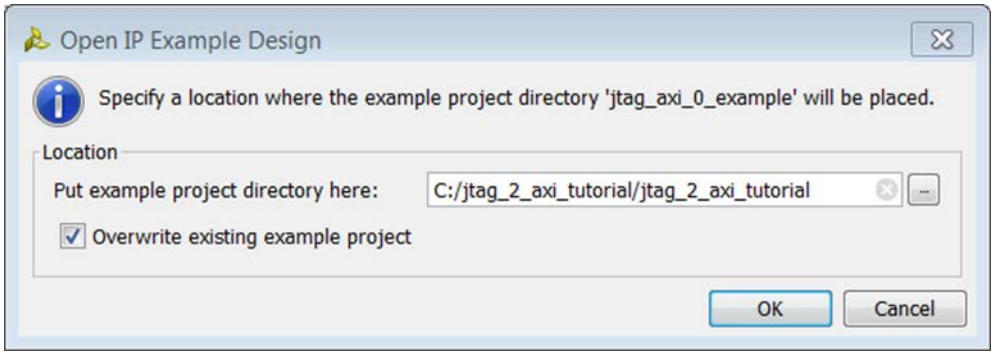


Figure 144: Open IP Example Design dialog

17. Open the `example_jtag_axi_0.v` file and notice that the jtag_axi_0 module is connected to an axi_bram_ctrl_0 (AXI-BRAM block memory) module.
18. In the example_jtag_axi_0.v file, add the following string to the beginning of the wire declaration for each `axi_*` signal from lines 72-108:

```
(* mark_debug *)
```

> ***Note**: Do not put `mark_debug` on the `axi_aclk` signal since this might result in Vivado Synthesis adding a LUT1 to the clock path, which could possibly cause you to not meet timing.*

Lines 72-108 should look like this:

```
(* mark_debug *) wire [31:0]axi_araddr;
(* mark_debug *) wire [1:0]axi_arburst;
(* mark_debug *) wire [3:0]axi_arcache;
(* mark_debug *) wire [0 :0]axi_arid;
(* mark_debug *) wire [7:0]axi_arlen;
(* mark_debug *) wire axi_arlock;
(* mark_debug *) wire [2:0]axi_arprot;
(* mark_debug *) wire [3:0]axi_arqos;
(* mark_debug *) wire axi_arready;
(* mark_debug *) wire [2:0]axi_arsize;
(* mark_debug *) wire axi_arvalid;
(* mark_debug *) wire [31:0]axi_awaddr;
(* mark_debug *) wire [1:0]axi_awburst;
(* mark_debug *) wire [3:0]axi_awcache;
(* mark_debug *) wire [0 :0]axi_awid;
(* mark_debug *) wire [7:0]axi_awlen;
(* mark_debug *) wire [7:0]axi_awlen;
(* mark_debug *) wire axi_awlock;
(* mark_debug *) wire [2:0]axi_awprot;
(* mark_debug *) wire [3:0]axi_awqos;
(* mark_debug *) wire axi_awready;
(* mark_debug *) wire [2:0]axi_awsize;
(* mark_debug *) wire axi_awvalid;
(* mark_debug *) wire [0 :0]axi_bid;
(* mark_debug *) wire axi_bready;
(* mark_debug *) wire [1:0]axi_bresp;
```

```
(* mark_debug *) wire axi_bvalid;
(* mark_debug *) wire [31 :0]axi_rdata;
(* mark_debug *) wire [0 :0]axi_rid;
(* mark_debug *) wire axi_rlast;
(* mark_debug *) wire axi_rready;
(* mark_debug *) wire [1:0]axi_rresp;
(* mark_debug *) wire axi_rvalid;
(* mark_debug *) wire [31 :0]axi_wdata;
(* mark_debug *) wire axi_wlast;
(* mark_debug *) wire axi_wready;
(* mark_debug *) wire [3 :0]axi_wstrb;
(* mark_debug *) wire axi_wvalid;
```

19. Save changes to `example_jtag_axi_o.v` file.
20. In the Flow Navigator on the left side of the Vivado window, click **Run Synthesis**.
21. Open the synthesized design by selecting **Open Synthesized Design** and clicking **OK**.

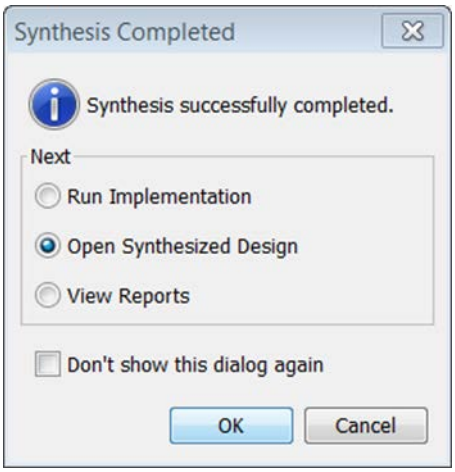


Figure 145: Open Synthesized Design

22. Once the synthesized design opens, do the following:

    a. Select the **Debug** layout in the main toolbar of the Vivado IDE.

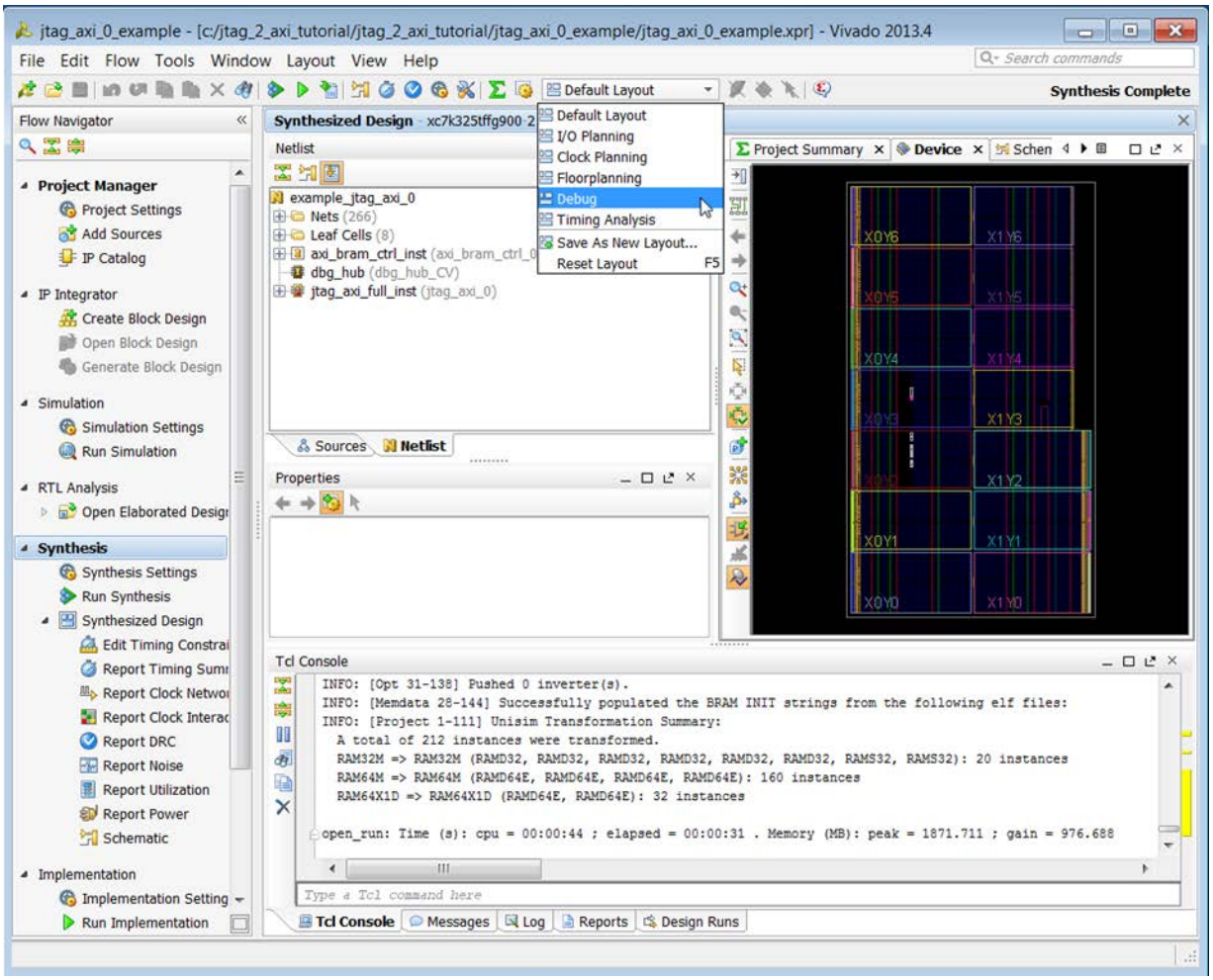


Figure 146: Debug Layout in the Vivado IDE toolbar

b. Select the **Debug** window near the bottom of the Vivado IDE.

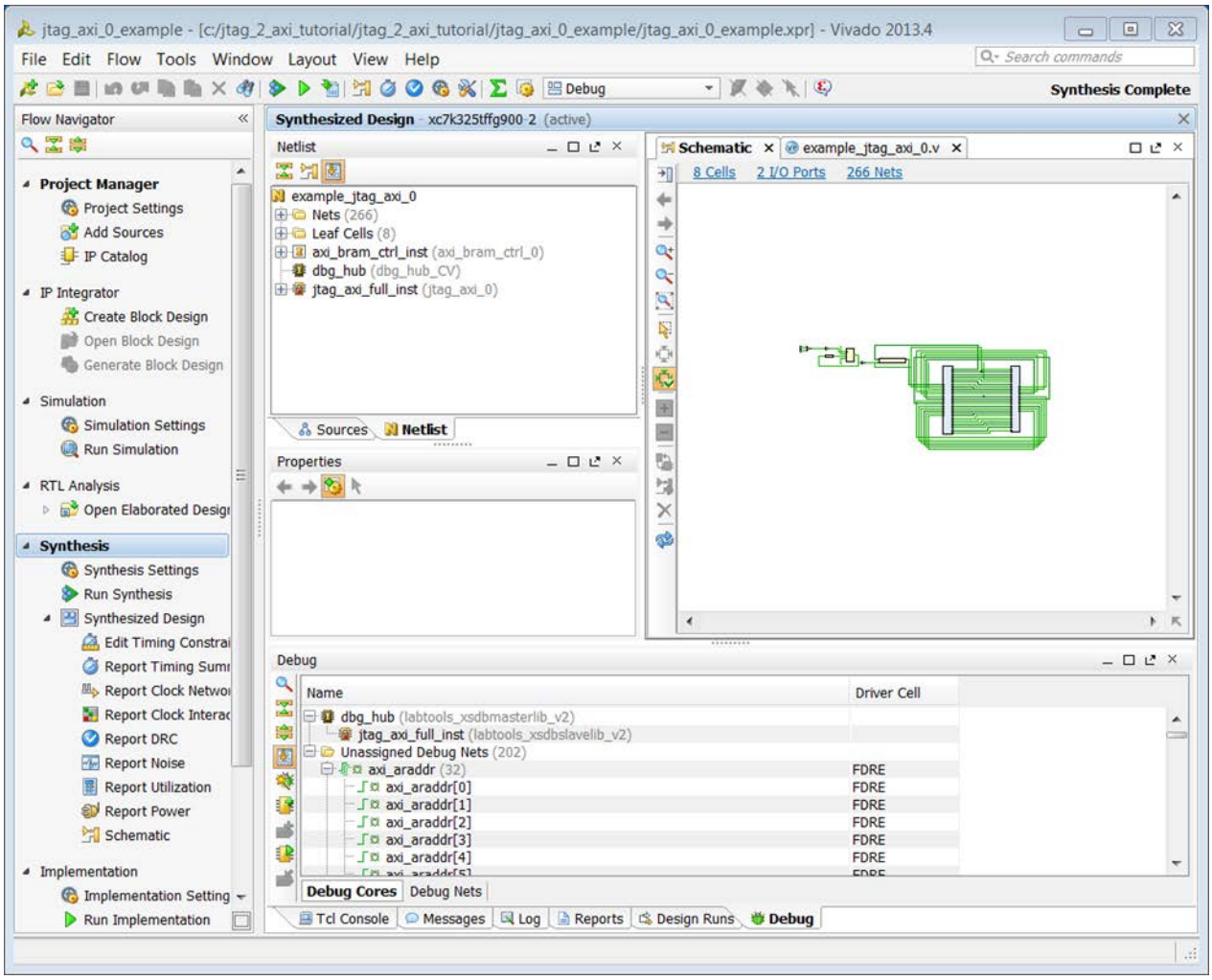


Figure 147: Debug window in the Vivado IDE

c. Click the **Set up Debug** toolbar button to launch the **Set up Debug** wizard.

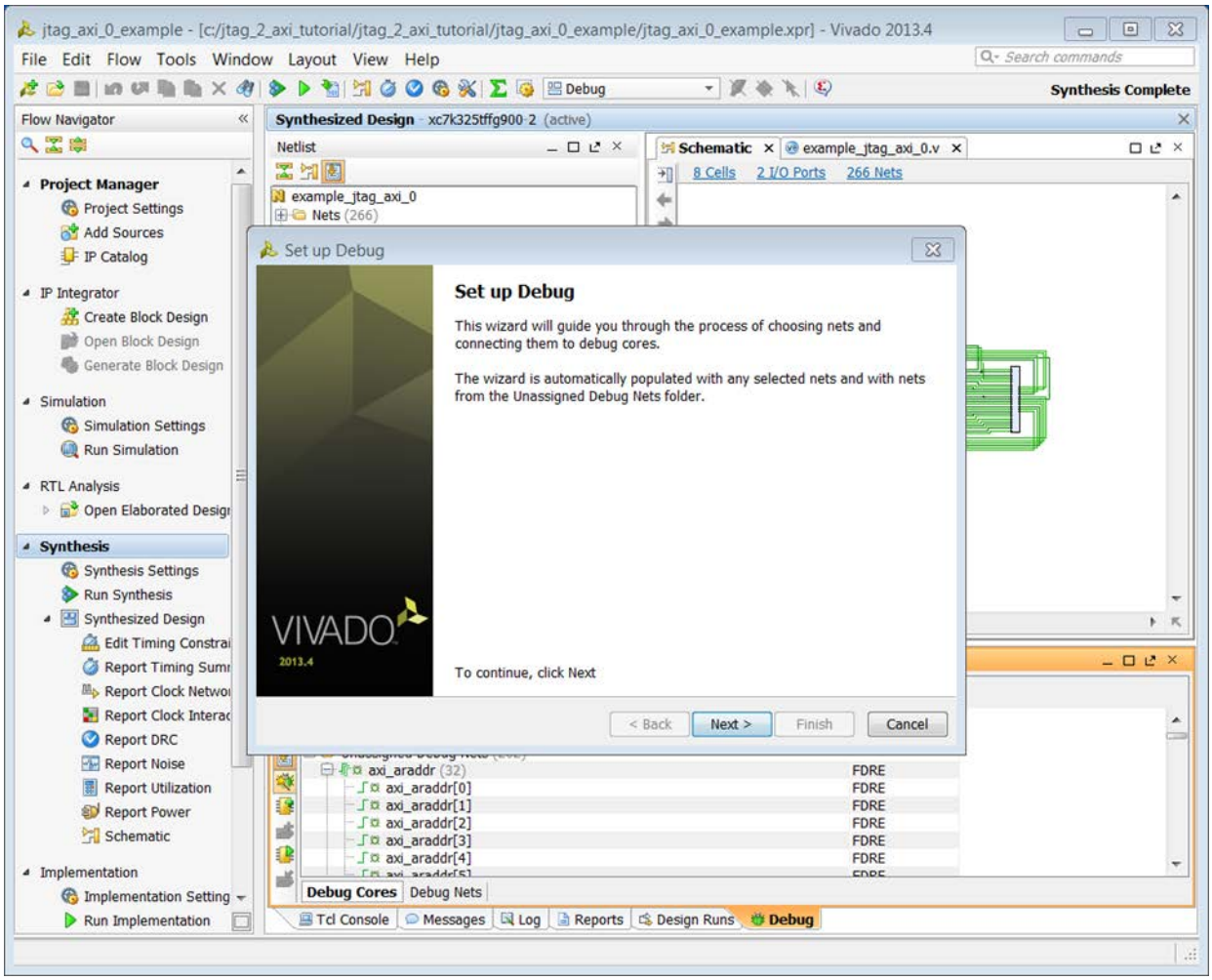


Figure 148: Set up Debug Wizard

23. Once the **Set up Debug** wizard pops up, click **Next**.

24. In the next page of the **Setup Debug** wizard, note that some of the nets that you would like to debug have no detectable clock domains selected. Click the **more info** link in the message banner.

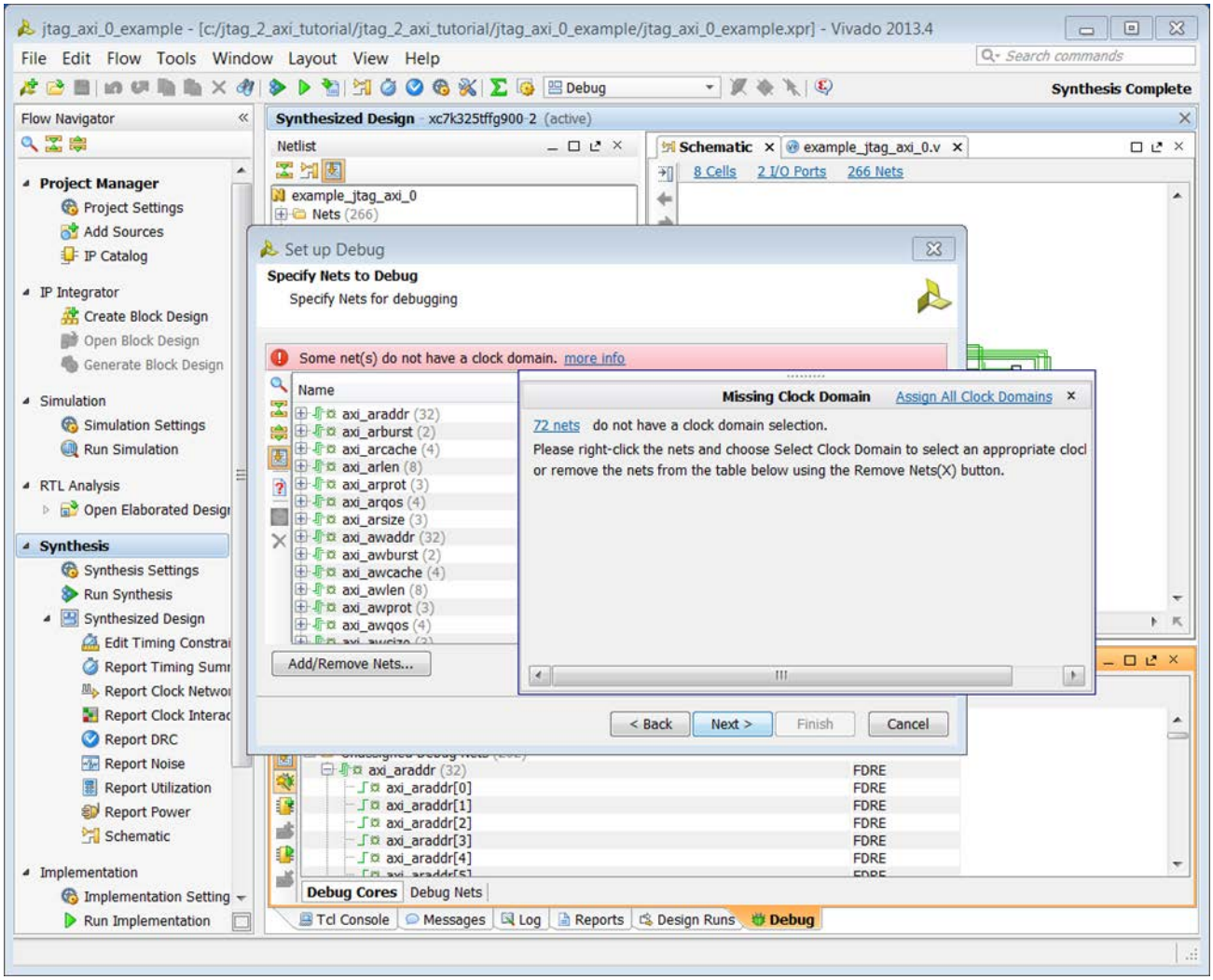


Figure 149: Missing Clock Domain dialog

25. In the resulting pop-up, click **Assign All Clock Domains**.

26. In the resulting pop-up, select the **aclk** clock net, then click **OK**.

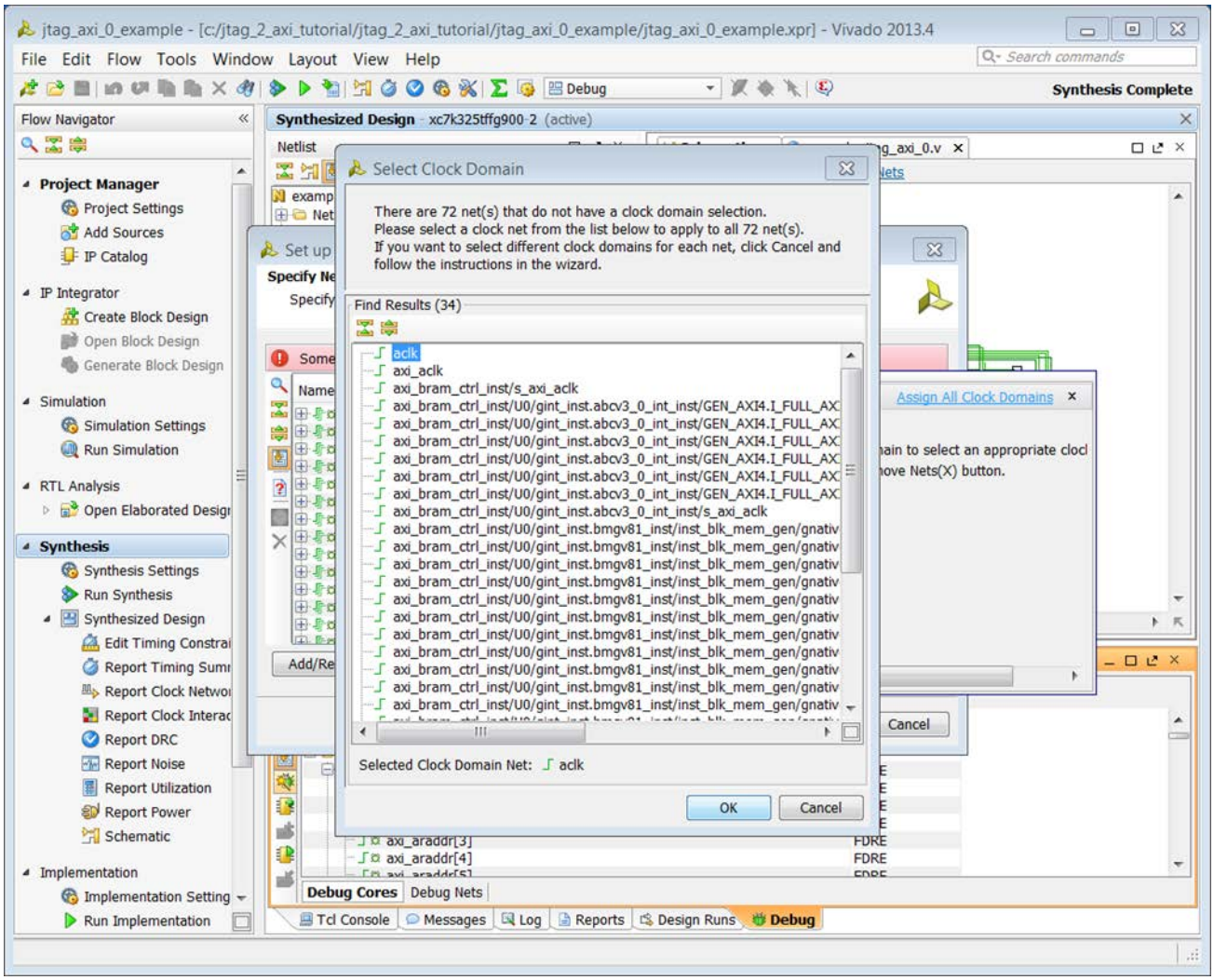


Figure 150: Select Clock Domain dialog

27. Observe that all of the nets now have an assigned clock domain. Click **Next**.

28. In the **Trigger and Storage Settings** page, ensure that **Advanced Trigger** and **Capture Control** are selected. Click **Next**.

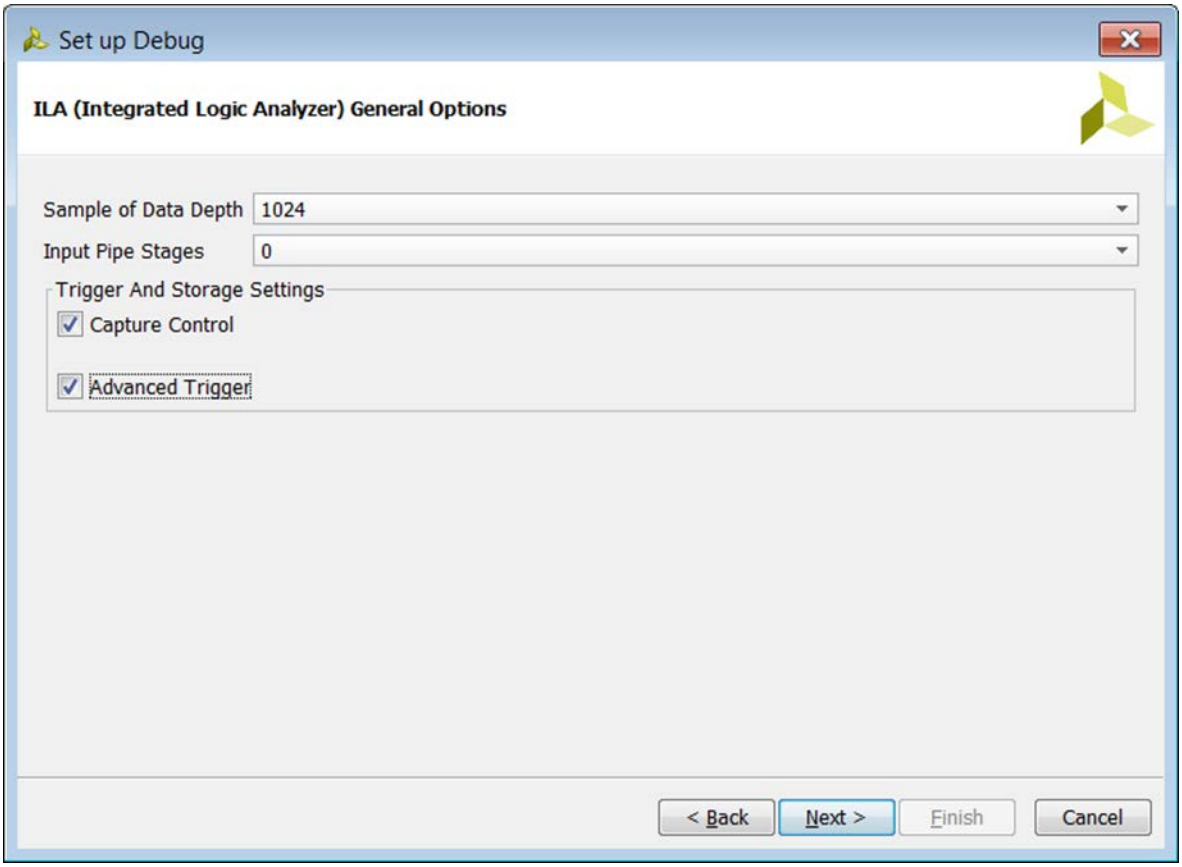


Figure 151: Trigger and Capture Modes page

29. Click **Finish**.

***Note**: See that the ILA core was inserted and attached to the dbg_hub core.*

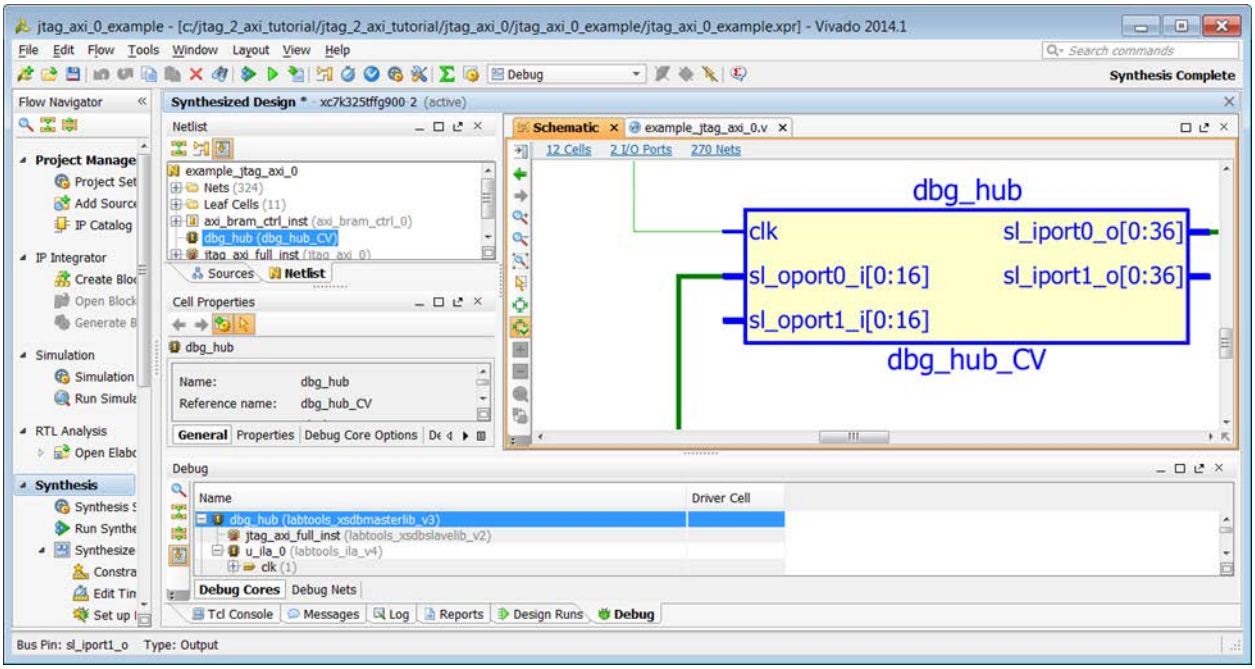


Figure 152: ILA core inserted into the design

30. Save the constraints by clicking **Save**.

31. The insertion of debug cores and changing of properties on those debug cores adds constraints to your target XDC constraint file. This modification of your target constraints file currently sets your synthesis out of date. You can force the design up to date by clicking **more info** in the upper-right corner of the Vivado IDE, then clicking **Force up-to-date**.

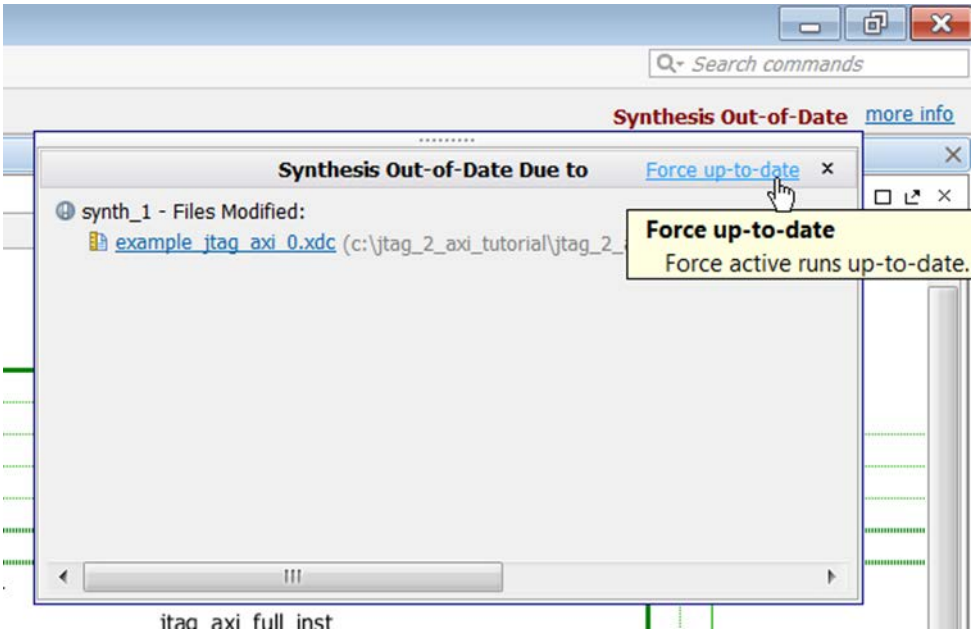


Figure 153: Forcing Synthesis up-to-date

32. In the **Flow Navigator** on the left side of the Vivado IDE, click **Generate Bitstream**.
33. Click **Yes** to implement the design.
34. Wait until the Vivado status shows **write_bitstream complete**.

35. In the **Bitstream Generation Completed** dialog box, select **Open Hardware Manager** and click **OK**.

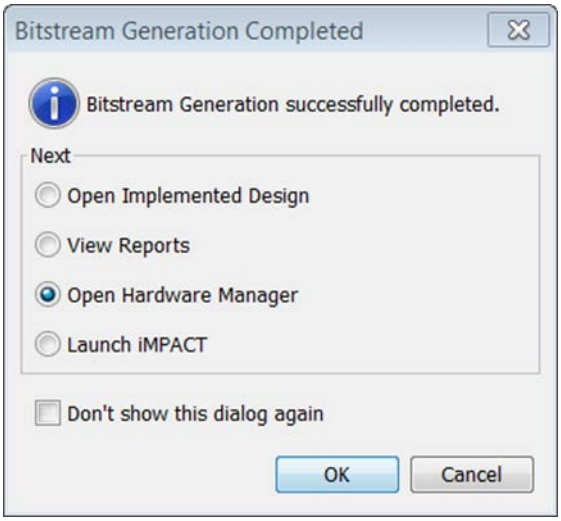


Figure 154: Open Hardware Manager

Step 2: Program the KC705 Board and Interact with the JTAG to AXI Master Core

1. Connect your KC705 board's USB-JTAG interface to a machine that has Vivado IDE and cable drivers installed on it and power up the board.
2. The **Hardware Manager** window opens. Click **Open a new hardware target**. The **Open New Hardware Target** dialog box opens.

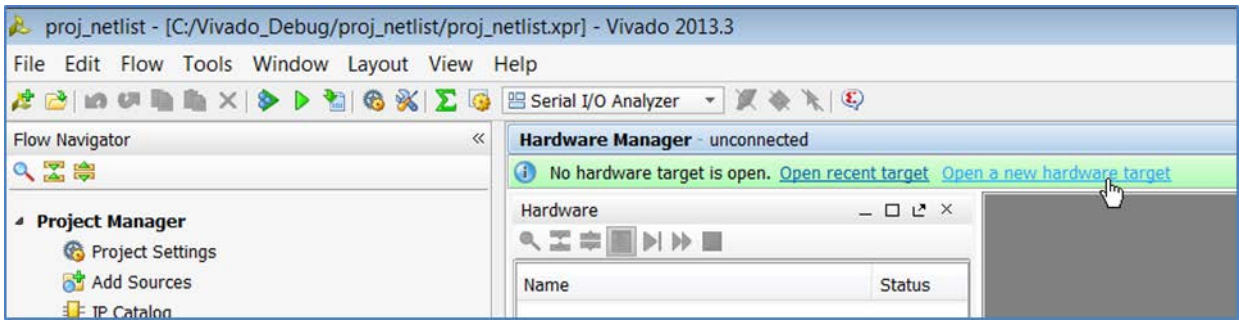


Figure 155: Connect to a Hardware Target

3. In the **Connect to** field choose **Local server** and click **Next**.

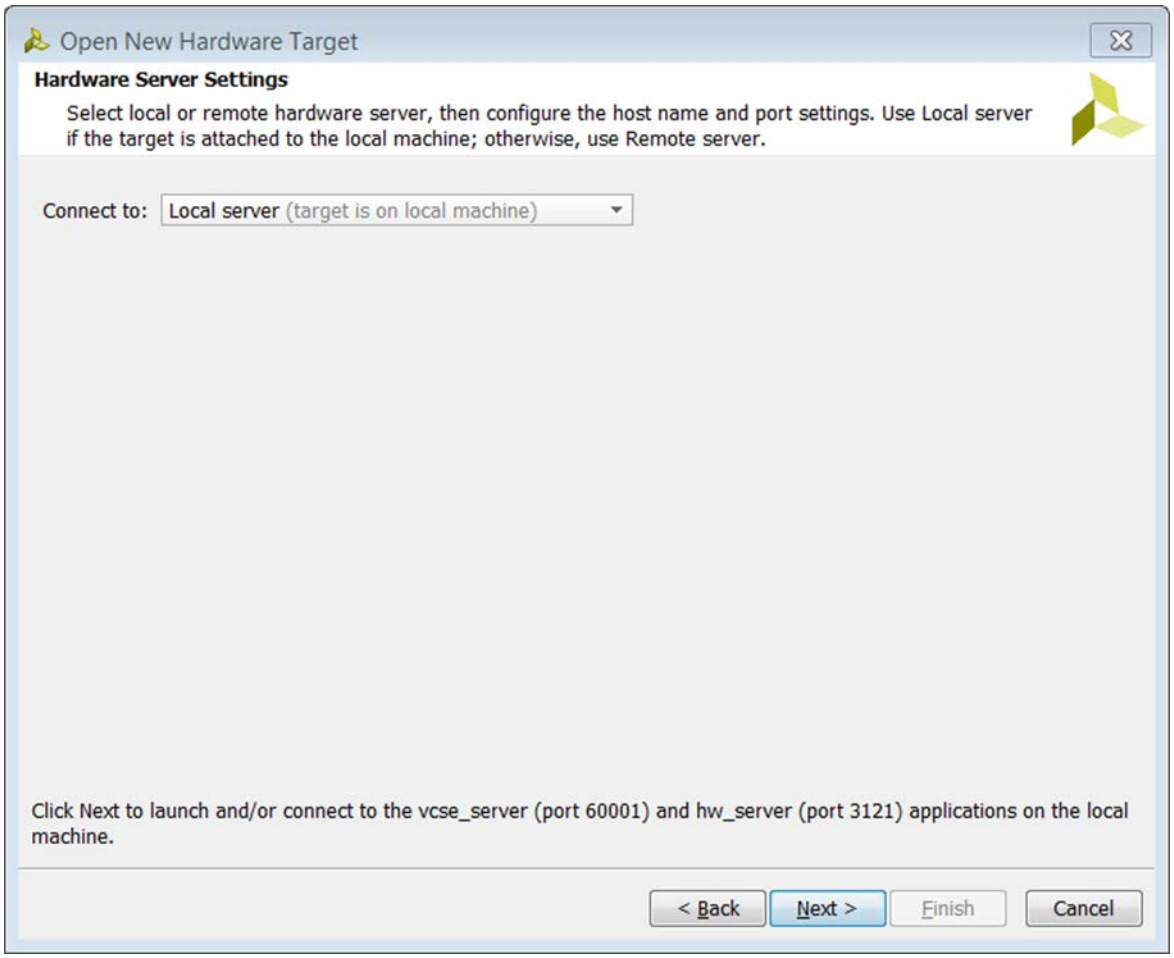

Open New Hardware Target
Hardware Server Settings
Select local or remote hardware server, then configure the host name and port settings. Use Local server if the target is attached to the local machine; otherwise, use Remote server.
Connect to:
Local server (target is on local machine)
Click Next to launch and/or connect to the vcse_server (port 60001) and hw_server (port 3121) applications on the local machine.
< Back
Next >
Finish
Cancel

Figure 156: Hardware Server Name

***Note**: Depending on your connection speed, this may take about 10 to 15 seconds.*

4. If there is more than one target connected to the hardware serve, you will see multiple entries in the **Select Hardware Target** page. In this tutorial, there is only one target as shown in the following figure. Click **Next**.

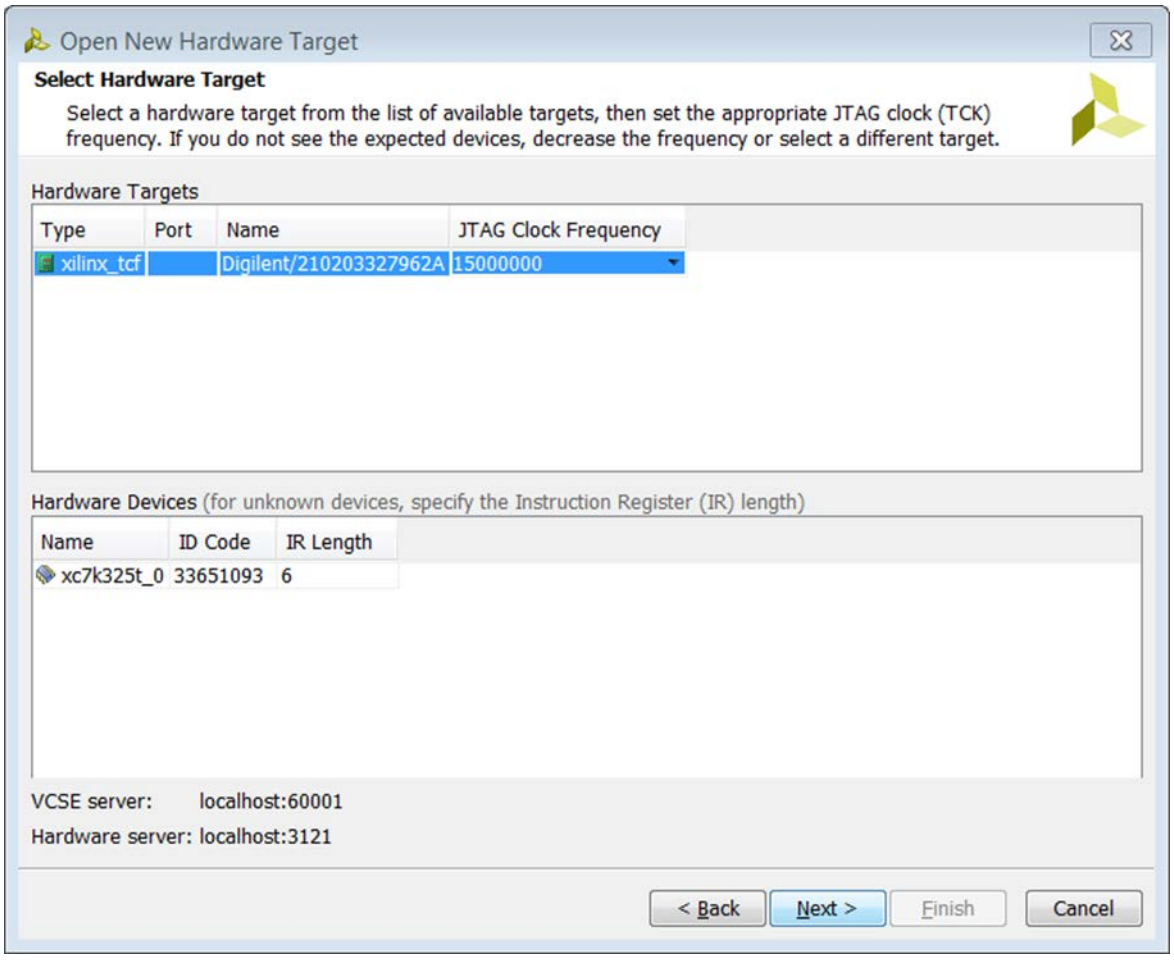


Figure 157: Select Hardware Target

5. Leave these settings at their default values as shown. Click **Next**.

6. In the **Open Hardware Target Summary** page, click **Finish** as shown in the following figure.

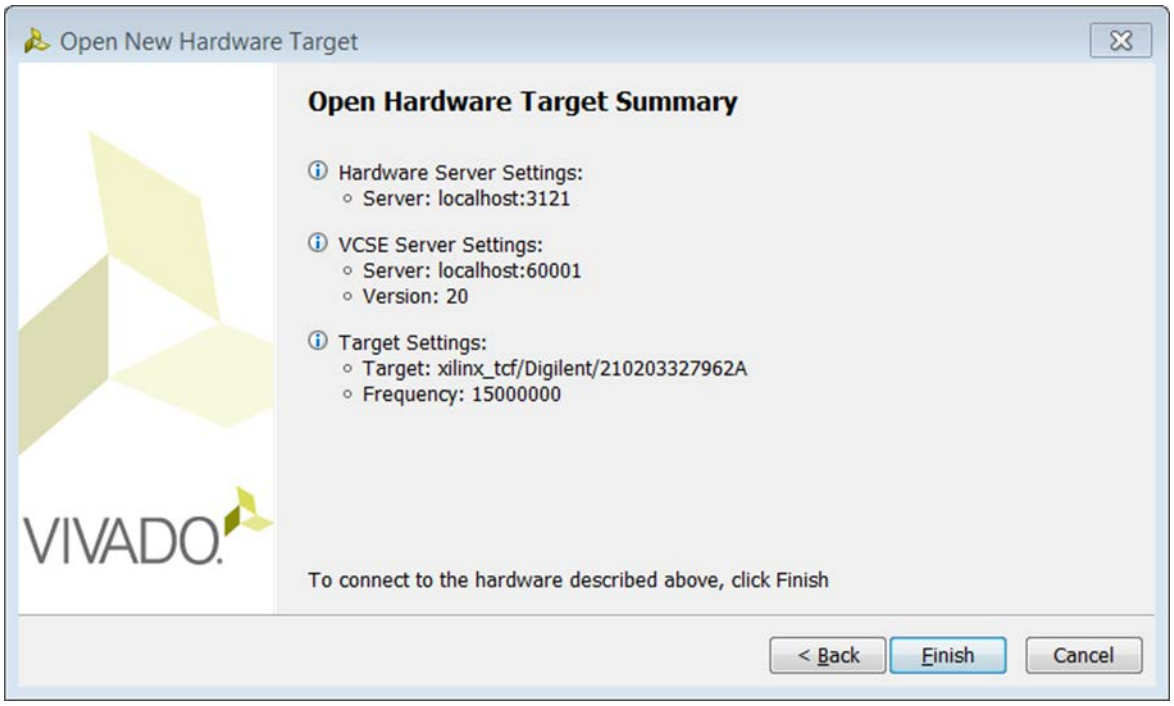


Figure 159: Open Hardware Summary

7. Wait for the connection to the hardware to complete. The dialog in the following figure appears while hardware is connecting.

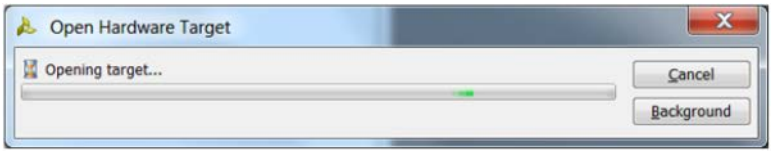
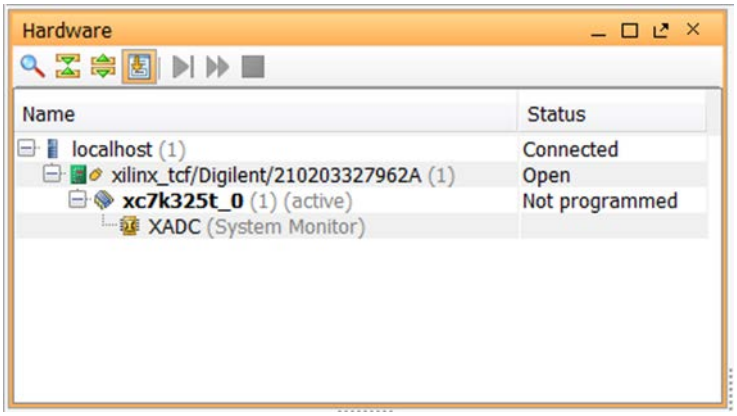


Figure 160: Open Hardware Target

Once the connection to the hardware target is made, the dialog shown in the following figure appears.

***Note**: The **Hardware** tab in the **Debug** view shows the hardware target and XC7K325T device that was detected in the JTAG chain.*



8. Next, program the XC7K325T device using the `.bit` bitstream file that was created previously by right-clicking the **XC7K325T** device and selecting **Program Device** as shown in the following figure.

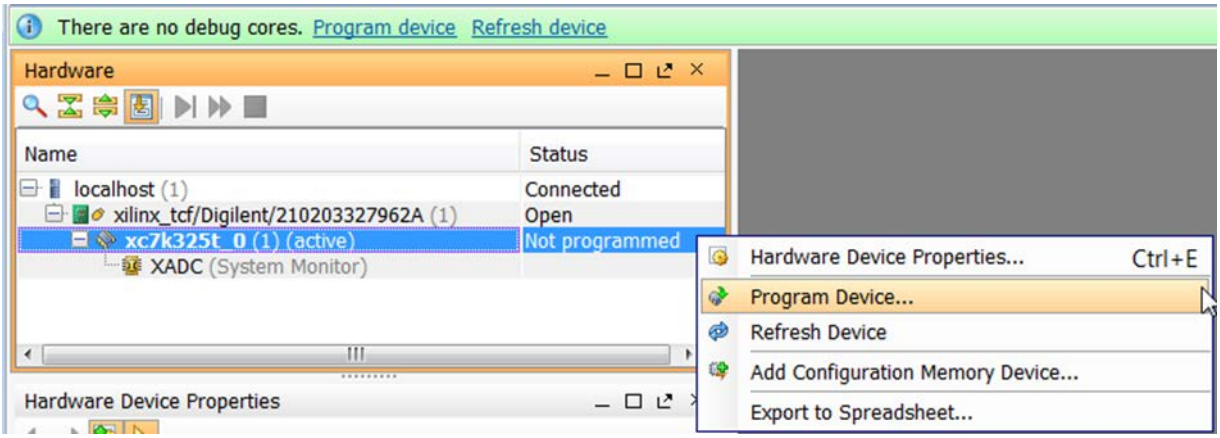


Figure 162: Program Active Target Hardware

9. In the **Program Device** dialog box verify that the `.bit` file is correct for the lab that you are working on. Click the **OK** to program the device.

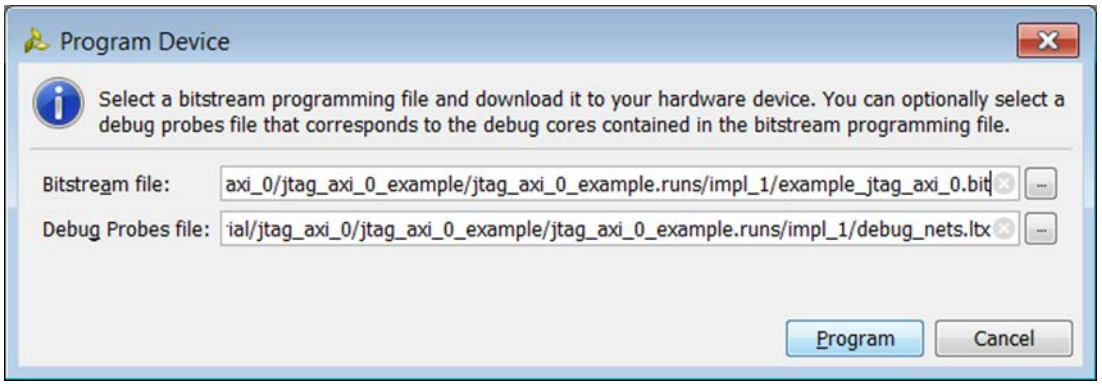

Program Device
Select a bitstream programming file and download it to your hardware device. You can optionally select a debug probes file that corresponds to the debug cores contained in the bitstream programming file.
Bitstream file:
axi_0/jtag_axi_0_example/jtag_axi_0_example.runs/impl_1/example_jtag_axi_0.bit
Debug Probes file:
ial/jtag_axi_0/jtag_axi_0_example/jtag_axi_0_example.runs/impl_1/debug_nets.ltx
Program
Cancel

Figure 163: Select Bitstream file to download

***Note**: Wait for the program device operation to complete. This may take few minutes.*

10. Verify that the JTAG to AXI Master and ILA cores are detected by locating the **hw_axi_1** and **hw_ila_1** instances in the **Hardware Manager** window.

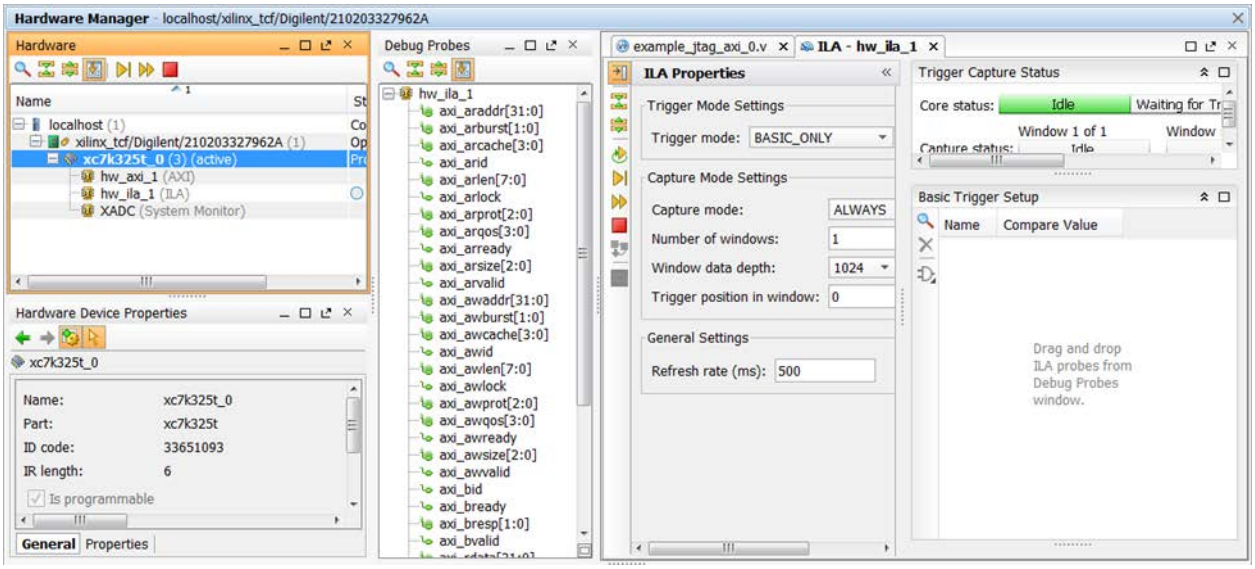


Figure 164: ILA core instances in the Hardware window

11. You can communicate with the JTAG to AXI Master core with Tcl commands only. You can issue AXI read and write transactions using the **run_hw_axi** command. However, before issuing these transactions, it is important to reset the **JTAG to AXI Master** core. Because the **aresetn** input port of the `jtag_axi_0` core instance is not connected to anything, you need to use the following Tcl commands to reset the core:

    ```
    reset_hw_axi [get_hw_axis hw_axi_1]
    ```

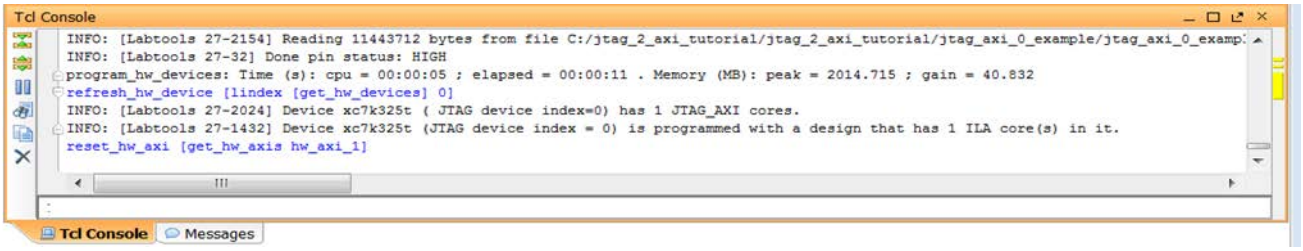

Tcl Console
INFO: [Labtools 27-2154] Reading 11443712 bytes from file C:/jtag_2_axi_tutorial/jtag_2_axi_tutorial/jtag_axi_0_example/jtag_axi_0_examp
INFO: [Labtools 27-32] Done pin status: HIGH
program_hw_devices: Time (s): cpu = 00:00:05 ; elapsed = 00:00:11 . Memory (MB): peak = 2014.715 ; gain = 40.832
refresh_hw_device [lindex [get_hw_devices] 0]
INFO: [Labtools 27-2024] Device xc7k325t (JTAG device index=0) has 1 JTAG_AXI cores.
INFO: [Labtools 27-1432] Device xc7k325t (JTAG device index = 0) is programmed with a design that has 1 ILA core(s) in it.
reset_hw_axi [get_hw_axis hw_axi_1]
Tcl Console
Messages

Figure 165: Reset JTAG to AXI core

12. The next step is to create a 4-word AXI burst transaction to write to the first four locations of the BRAM:

    ```
    set wt [create_hw_axi_txn write_txn [get_hw_axis hw_axi_1] -type WRITE -address
    00000000 -len 128 -data {44444444_33333333_22222222_11111111}]
    ```

    where:

    - "write_txn" is the name of the transaction
    - "[get_hw_axis hw_axi_1]" returns the hw_axi_1 object
    - "-address 00000000" is the start address
    - "-len 4" sets the AXI burst length to 128 words
    - "-data {44444444_33333333_22222222_11111111}" is the data to be written.

    ***Note**: The data direction is MSB to the left (i.e., address 3) and LSB to the right (i.e., address 0). Also note that the data will be repeated from the LSB to the MSB to fill up the entire burst.*

13. The next step is to set up a 128-word AXI burst transaction to read the contents of the first four locations of the AXI-BRAM core:

```
set rt [create_hw_axi_txn read_txn [get_hw_axis hw_axi_1] -type READ -address 00000000 -len 128]
```

    where:

    - `read_txn` is the name of the transaction
    - `[get_hw_axis hw_axi_1]` returns the hw_axi_1 object
    - `-address 00000000` is the start address
    - `-len 128` sets the AXI burst length to 4 words

14. After creating the transaction, you can run it as a write transaction using the run_hw_axi command:

```
run_hw_axi $wt
```

This command should return the following:

```
INFO: [Labtools 27-147]  vcse_server: WRITE DATA is :
44444444333333332222222211111111...
```

15. After creating the transaction, you can run it as a read transaction using the `run_hw_axi` command:

```
run_hw_axi $rt
```

This command should return the following:

```
INFO: [Labtools 27-147] vcse_server: READ DATA is :
44444444333333332222222211111111...
```

Step 3: Using ILA 3.0 Advanced Trigger Feature to Trigger on an AXI Read Transaction

1. Open the ILA core's dashboard by right-clicking **hw_ila_1** in the **Hardware Manager** window and selecting **Open Dashboard**.

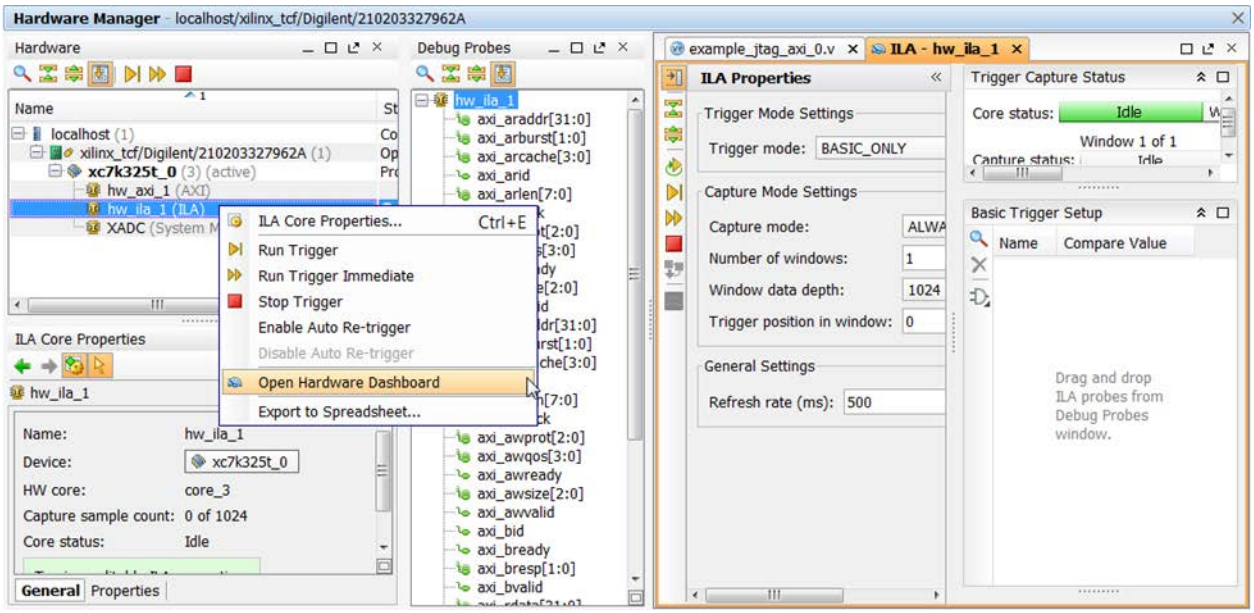


Figure 166: Opening the ILA Dashboard

2. In the **ILA – hw_ila_1** dashboard, locate **the Trigger Mode Settings** area and **set Trigger mode** to **ADVANCED_ONLY**.

3. In the **Capture Mode Settings** area set the **Trigger position** to **512**.
4. In the **Trigger State Machine** area click the **Create new trigger state machine** link.

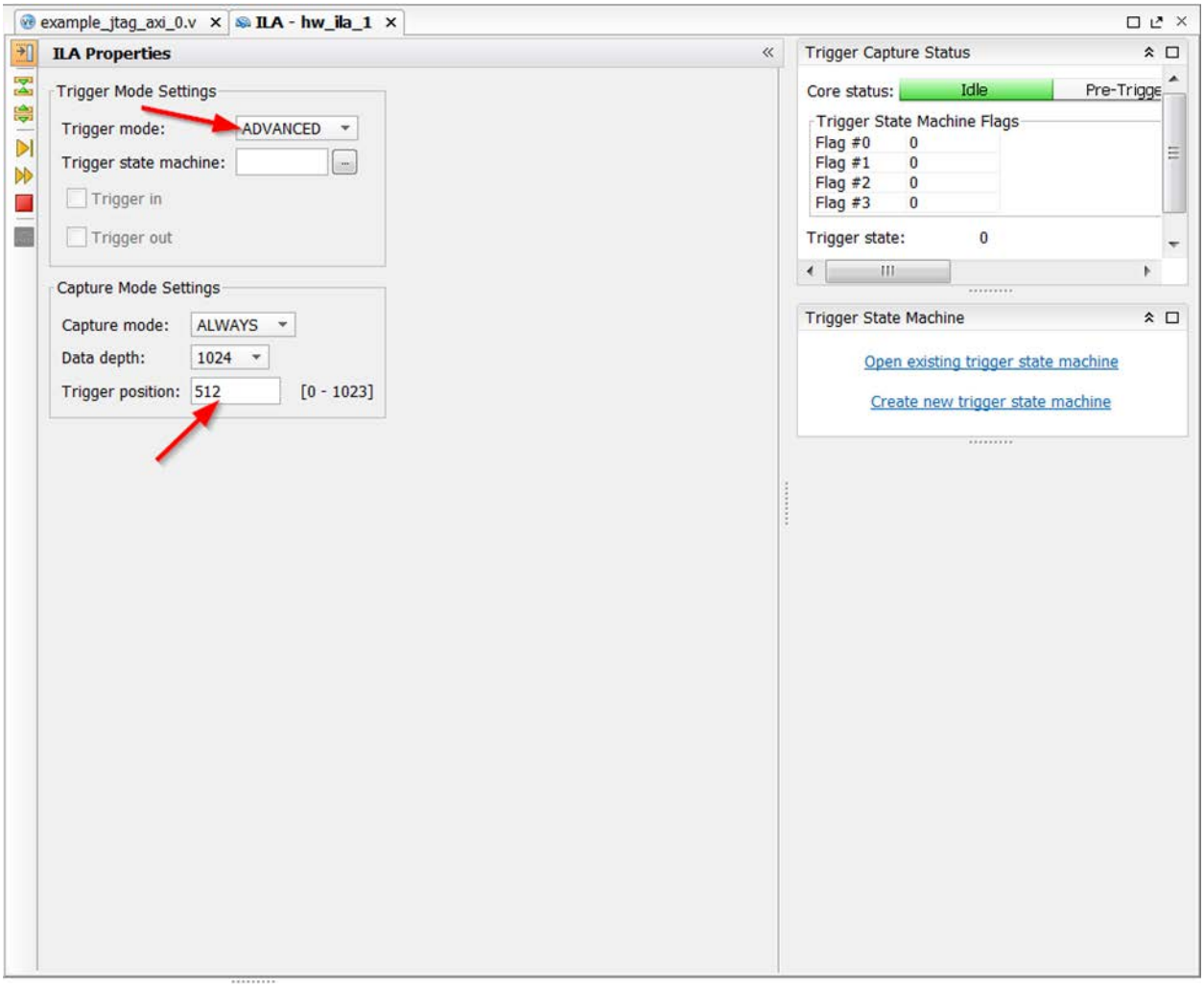


Figure 167: Setting Trigger mode to ADVANCED and Trigger Position to 512 in the ILA Dashboard

5. In the **New Trigger State Machine File** dialog box set the name of the state machine script to **txns.tsm**.

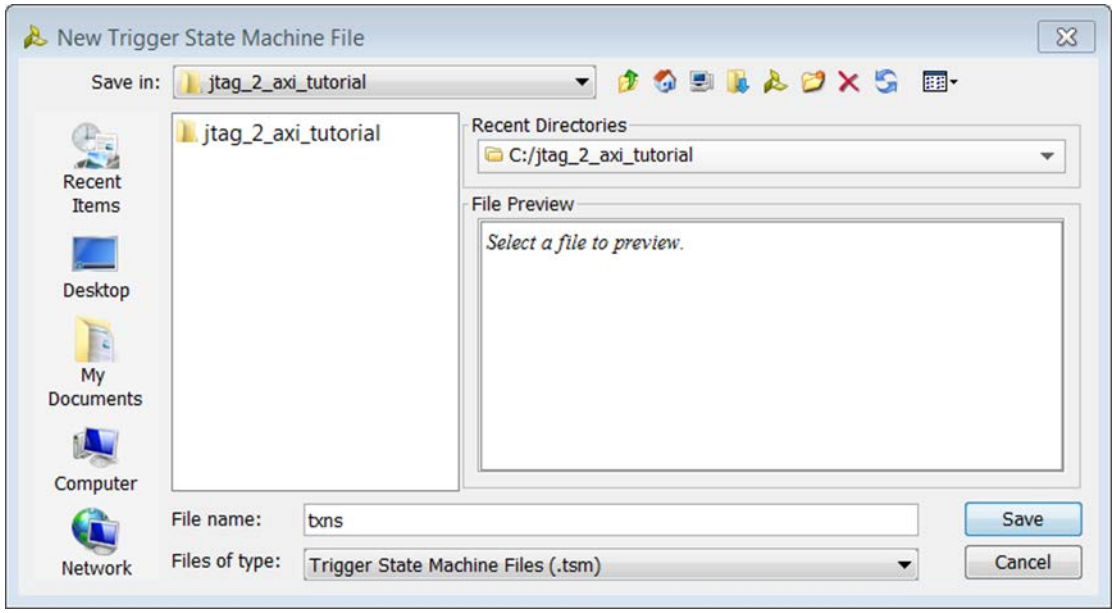


Figure 168: Creating a new Trigger State Machine script

6. A basic template of the trigger state machine script is displayed in the Trigger State Machine gadget. Expand the trigger state machine gadget in the ILA dashboard. Copy the script below after line 17 of the state machine script and save the file.

```
#
# The "wait_for_arvalid" state is used to detect the start
# of the read address phase of the AXI transaction which
# is indicated by the axi_arvalid signal equal to '1'
#
state wait_for_arvalid:
    if (axi_arvalid == 1'b1) then
      goto wait_for_rready;
    else
      goto wait_for_arvalid;
    endif
#
# The "wait_for_rready" state is used to detect the start
# of the read data phase of the AXI transaction which
# is indicated by the axi_rready signal equal to '1'
#
state wait_for_rready:
  if (axi_rready == 1'b1) then
    goto wait_for_rlast;
  else
    goto wait_for_rready;
  endif
```

```
#
# The "wait_for_rlast" state is used to detect the end
# of the read data phase of the AXI transaction which
# is indicated by the axi_rlast signal equal to '1'.
# Once the end of the data phase is detected, the ILA core
# will trigger.
#
state wait_for_rlast:
  if (axi_rlast == 1'b1) then
    trigger;
  else
    goto wait_for_rlast;
  endif
```

***Note**: The state machine is used to detect the various phases of an AXI read transaction:*

- Beginning of the read address phase.
- Beginning of the read data phase.
- End of the read data phase.

7. Arm the trigger of the ILA by right-clicking the **hw_ila_1** core in the **Hardware Manager** window and selecting **Run Trigger**.

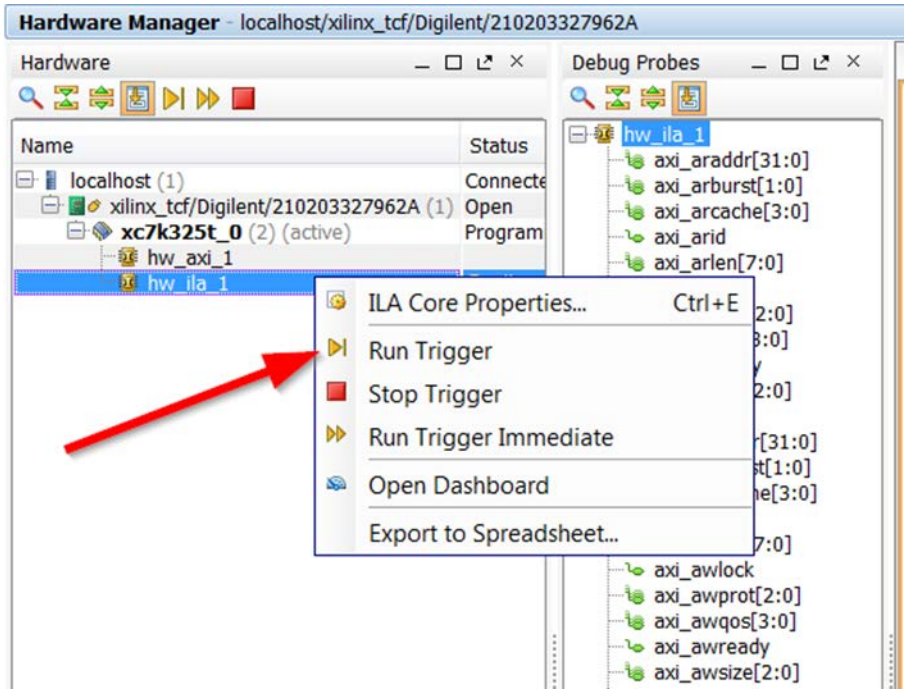


Figure 169: Run Trigger

8. In the **Trigger Capture Status** window, note that the ILA core is waiting for the trigger to occur, and that the trigger state machine is in the **wait_for_a_valid** state. Note that the pre-trigger capture of 512 samples has completed successfully:

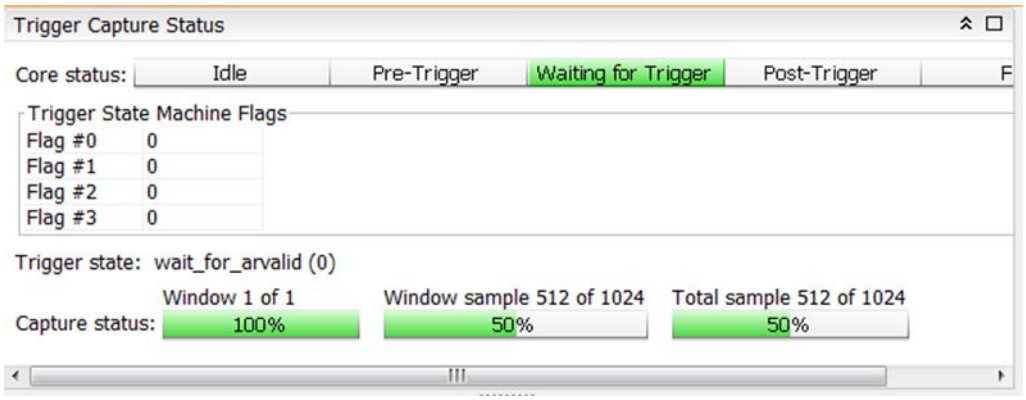


Figure 167: Trigger Capture Status window

9. In the Tcl console, run the read transaction that you set up in the previous section of this tutorial.

```
run_hw_axi $rt
```

***Note**: The ILA core has triggered and the trigger mark is on the sample where the `axi_rlast` signal is equal to '1', just as the trigger state machine program intended.*

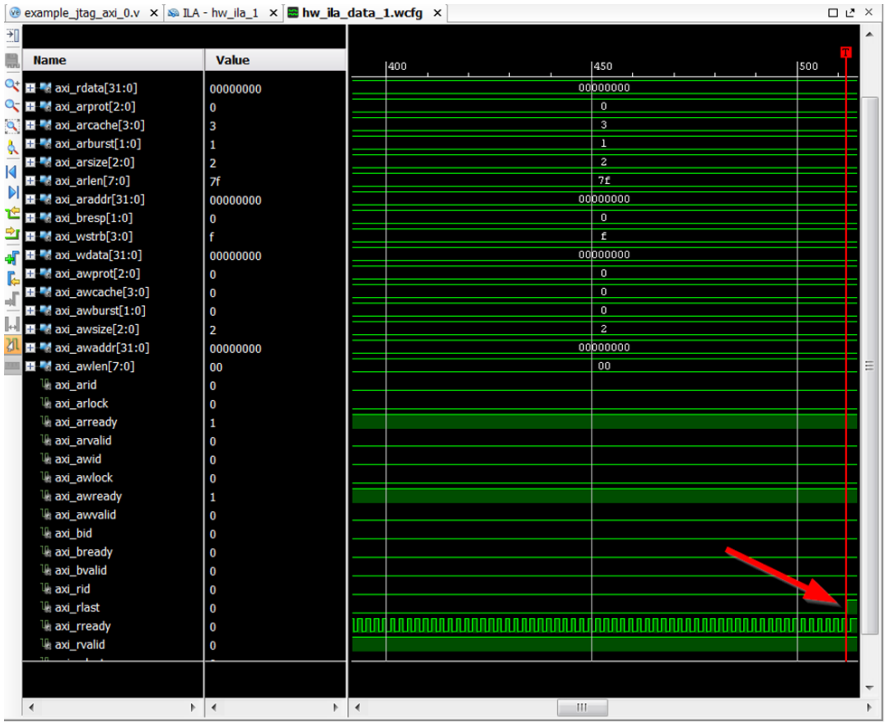


Figure 170: Waveform window

Please Read: Important Legal Notices

The information disclosed to you hereunder (the “Materials”) is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx’s limited warranty, please refer to Xilinx’s Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx’s Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos.

© Copyright 2014 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.